



**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

BRNO UNIVERSITY OF TECHNOLOGY

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

FACULTY OF INFORMATION TECHNOLOGY

**ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ**

DEPARTMENT OF COMPUTER SYSTEMS

**SIMULACE LOMOVÉ ZKOUŠKY VE STAVEBNICTVÍ**

SIMULATION OF FRACTURE TESTS IN CIVIL ENGINEERING

**DIPLOMOVÁ PRÁCE**

MASTER'S THESIS

**AUTOR PRÁCE**

AUTHOR

**Bc. GABRIEL BORDOVSKÝ**

**VEDOUCÍ PRÁCE**

SUPERVISOR

**Ing. JIŘÍ JAROŠ, Ph.D.**

**BRNO 2017**

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav počítačových systémů

Akademický rok 2016/2017

**Zadání diplomové práce**

Řešitel: **Bordovský Gabriel, Bc.**

Obor: Počítačové a vestavěné systémy

Téma: **Simulace lomové zkoušky ve stavebnictví**  
**Simulation of Fracture Tests in Civil Engineering**

Kategorie: Algoritmy a datové struktury

**Pokyny:**

1. Seznamte se s architekturou superpočítačů Anselm a Salomon.
2. Osvojte si programovací techniky a potřebné programové prostředky pro implementaci vysoce náročných aplikací na superpočítači.
3. Prostudujte simulační program PreCraS, analyzujte jeho výkonnost a identifikujte slabá místa.
4. Navrhněte možnosti akcelerace současné implementace.
5. Navržené úpravy implementujte.
6. Ověřte výkonnost vytvořené implementace a porovnejte ji s původním kódem.
7. Zhodnoťte dosažené výsledky a diskutujte přínos pro praxi.

**Literatura:**

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 4 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese

<http://www.fit.vutbr.cz/info/szz/>

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Jaroš Jiří, Ing., Ph.D., UPSY FIT VUT**

Datum zadání: 1. listopadu 2016

Datum odevzdání: 24. května 2017


**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

Fakulta informačních technologií

Ústav počítačových systémů a sítí

612 66 Brno, Božetěchova 2

L.S.



prof. Ing. Lukáš Sekanina, Ph.D.  
vedoucí ústavu

## Abstrakt

Tato práce optimalizuje program pro simulaci lomové zkoušky ve stavebnictví. Simulace je využívána k validaci lomových vlastností materiálu používaných při rekonstrukci historických budov. Názorně jsou prezentovány možnosti efektivnějšího využití procesoru při zachování kvality výsledků. V práci jsou analyzovány jednotlivé kroky simulace a následně navrženy možnosti optimalizace kritických úseku pomocí vektorizace či paralelizace. Techniky a postupy použité v této práci mohou být aplikovány na obdobné výpočty a tím výrazně zkrátit dobu potřebnou k výpočtu. Čas výpočtu prototypu byl přes 7,7 hodiny. Optimalizovaná verze zvládá sekvenčně stejný výpočet za 2,1 hodiny nebo paralelně na osmi jádrech za 21 minut. Oproti původní verzi je tak optimalizovaná paralelní verze 21-krát rychlejší.

## Abstract

In this thesis, a program for fracture test in civil engineering has been optimized. The simulation is used for a validation of the fracture characteristics for blocks of construct material used for historic buildings reconstructure. This thesis illustrates the possibilities of an effective usage of the processor's potential without the loss of the output quality. The individual parts of the simulation are analyzed and this thesis proposes for the critical sections some possible optimizations such as vectorization or parallel processing. The techniques used in this thesis may be used on similar computing problems and help shorten the required runtime. The prototype of the simulation was able to process the simulation in 7.7 hours. Optimized version is capable to process the same simulation in 2.1 hours on one core or 21 minutes on eight cores. The parallel optimized version is 21 times faster than the prototype.

## Klíčová slova

Paralelizace, Vektorizace, CUDA, Optimalizace, Kvazikřehké materiály, Zkouška třibodovým ohbem, Metoda konečných prvku

## Keywords

Parallelism, Vectorization, CUDA, Optimization, Quasi-brittle materials, fracture test by three point flexion, Finite element method

## Citace

BORDOVSKÝ, Gabriel. *Simulace lomové zkoušky ve stavebnictví*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Jaroš Jiří.

# Simulace lomové zkoušky ve stavebnictví

## Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jiřího Jaroše, Ph.D. Další informace mi poskytl autor prototypu, Ing. Jan Bedáň z Ústavu stavební mechaniky FAST VUT v Brně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....  
Gabriel Bordovský  
23. května 2017

## Poděkování

Děkuji Ing. Jiřímu Jaroši Ph.D. za pomoc s vypracováním práce. Tato práce byla podpořována Ministerstvem školství, mládeže a tělovýchovy z projektu "IT4Innovations National Supercomputing Center – LM2015070“ pro podporu velkých výzkumných infrastruktur, experimentální vývoj a inovativní projekty.

# Obsah

<b>1</b>	<b>Úvod</b>	<b>3</b>
<b>2</b>	<b>Zkouška třibodovým ohybem</b>	<b>4</b>
2.1	Druhy materiálu . . . . .	4
2.2	Průběh zkoušky . . . . .	4
<b>3</b>	<b>Simulace lomové zkoušky</b>	<b>7</b>
3.1	Motivace pro simulaci . . . . .	7
3.2	Vstupní model trámce . . . . .	8
3.3	Příprava dat pro simulaci . . . . .	9
3.4	Běh simulace . . . . .	11
3.5	Algoritmus pro výpočet formování trhliny . . . . .	12
<b>4</b>	<b>Analýza původní simulace</b>	<b>13</b>
4.1	Popis implementace . . . . .	13
4.2	Využití procesoru . . . . .	15
4.3	Slabé místo . . . . .	16
4.4	Vliv překladače . . . . .	16
4.5	I/O . . . . .	17
<b>5</b>	<b>Metody optimalizace</b>	<b>18</b>
5.1	Vektorizace - SIMD . . . . .	18
5.2	Zákony o paralelizaci . . . . .	19
5.3	Vláknový paralelizmus . . . . .	20
5.4	Aplikační rozhraní pro paralelizaci . . . . .	22
5.5	Grafické karty-CUDA . . . . .	24
5.6	Organizace dat v paměti . . . . .	25
<b>6</b>	<b>Optimalizace slabých míst</b>	<b>27</b>
6.1	I/O . . . . .	27
6.2	Paralelizace konečných elementů . . . . .	27
6.3	Zaměna Double za Float . . . . .	28
6.4	Vektorizace . . . . .	30
6.5	Rozbalení smyčky . . . . .	31
6.6	Odstranění režie tvorby vláken . . . . .	31

<b>7</b>	<b>Analyza výsledné verze</b>	<b>34</b>
7.1	Optimalizované využití procesoru . . . . .	34
7.2	Kritická sekce . . . . .	35
7.3	Škalovatelnost . . . . .	36
7.4	Závislost rychlosti na frekvenci procesoru . . . . .	37
7.5	Použití technologie Hyper-Threading . . . . .	39
7.6	Přesnost výsledků . . . . .	42
7.7	CUDA realizace . . . . .	42
<b>8</b>	<b>Závěr</b>	<b>44</b>
	<b>Literatura</b>	<b>46</b>
	<b>Přílohy</b>	<b>48</b>
	Seznam příloh . . . . .	49
<b>A</b>	<b>Obsah přiloženého paměťového média</b>	<b>50</b>

# Kapitola 1

## Úvod

Problémy lomové mechaniky se začaly vědeckou obcí řešit v polovině 20. století, a to z důvodu náhlých kolapsů konstrukcí. Roku 1920 byl publikován první pokus o definování vlivu vady konstrukcí na tahovou únostnost [13]. Byl zde použit první zákon termodynamiky k zavedení energetické rovnováhy. Daný přístup umožňoval spočítat koncentrační napětí na stěně v okolí eliptické trhliny ideálně křehkého materiálu. Následoval dlouhý výzkum, jehož výsledkem byl model použitelný pro ocel. Vzniklé odvětví se nazývá *lineární elastická lomová mechanika (LELM)*. Vznikly různé modifikace *LELM*, umožňující mimo jiné postihnout změkčení materiálu. Jedná se například o *model fiktivní trhliny*, *model kohezivní trhliny* nebo *model pásu trhlín*. Simulace popsaná v této práci používá právě model *kohezivní trhliny*.

Účel simulačního nástroje je validace předpokládaných materiálových vlastností lomu kvazikřehkých materiálu. Tyto parametry je potřeba určit u vzorků z historických budov, které potřebují materiál pro rekonstrukci. Výsledek zkoušky třibodým ohybem by měl odpovídat simulaci s adekvátním modelem. Při rozdílném výsledku simulace je potřeba parametry upravit a simulaci opakovat, dokud nedojde ke shodě. Přímé vyčtení parametrů ze zkoušky je obtížné, jelikož se jejich vliv zároveň prolíná.

Tato práce se zabývá metodou a funkčním prototypem simulace třibodového ohybu kvazikřehkých materiálů. Prototyp simulačního nástroje vznikl na Fakultě stavební Vysokého učení technického v Brně [10]. Problém simulace do jisté míry shrnuje tato publikace [19]. Problémem prototypu je dlouhý čas potřebný k výpočtu jedné simulace. Pro jeden vzorek je potřeba provést sérii několika simulací. Je proto potřeba prototyp optimalizovat, aby čas na jednu simulaci byl co nejkratší. Cílem analýzy vzorků je nalezení vhodných náhradních materiálu pro historické stavby. Díky rychlejší identifikaci vlastností materiálu mohou být rekonstrukce provedeny rychleji.

Cílem této práce proto je analyzovat daný prototyp a určit slabá místa výpočtu simulace. Pro tato místa i pro celou simulaci dále analyzovat různé techniky akcelerace a optimalizace. Mezi tyto techniky patří vektorizace, paralelizace, zpracování pomocí vláken a další. Vhodné optimalizace byly do kodu implementovány. Následně je prezentováno, jak je těmito změnami ovlivněn výsledný čas simulace oproti prototypu. Práce se zaměřuje převážně na řešení pomocí procesoru, ale jsou analyzovány i možnosti využití několika procesorů či grafických karet.

## Kapitola 2

# Zkouška třibodovým ohybem

Zkouška třibodým ohybem patří mezi destruktivní, testovaný vzorek je při zkoušce znehodnocen a nelze jej již znovu použít. Zkouška se také řadí mezi statické. V každý okamžik zkoušky je na vzorek působeno stále stejně. Z výsledků lze odvodit únavové parametry materiálu. Mezi dynamické nestatické zkoušky patří například nárazové zkoušky. Při nich se skokově změni síla působící na vzorek.

### 2.1 Druhy materiálu

U reálných materiálů existuje vztah mezi napětím a přetvořením. Diagram prezentující tento vztah lze získat právě lomovou zkouškou. Na Obrázku 2.1 jsou tři ideální průběhy pro materiály z následujících skupin [18]:

**Křehké** (Lineárně pružné): jakmile překročí určitou mez tlaku, pružné chování končí a napětí klesá k nule. Typickým příkladem je sklo.

**Pružně plastické**: po dosažení určité hranice zůstává napětí konstantní při rostoucí deformaci. Příkladem je ocel.

**Kvazikřehké**: maximálního napětí není dosaženo lineárně jako u předchozích dvou. Již před ním začne docházet k drobnému změkčení materiálu. Následně napětí nelineárně klesá se zvyšující se deformací. Toto chování mají kompozitní materiály jakými jsou beton (bez ocelových výztuží) nebo pískovec.

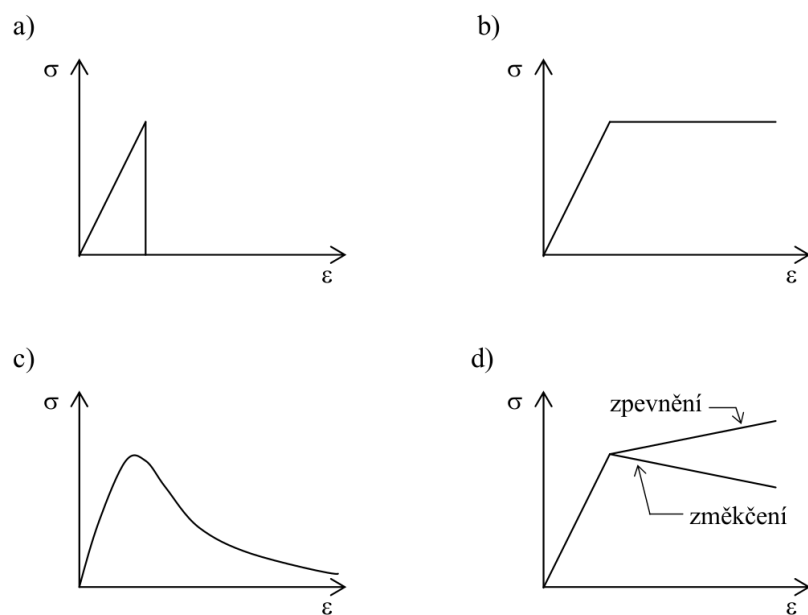
Z pohledu vztahu mezi napětím materiálu a jeho deformací jsou kvazikřehké materiály nejzajímavější. Výsledná křivka je značně komplikovanější než u zbylých dvou.

### 2.2 Průběh zkoušky

Vzorek materiálu, typicky označován jako trámec, je umístěn do zkušebního přístroje tak, že je zespodu podepřen dvěma podpěrami. Třetí bod zkoušky je umístěn ve středu tělesa nahoře. V tomto bodu je vzorek zatěžován konstantním ohybovým momentem síly až do úplného porušení vzorku. V odborné literatuře lze také nalézt, že zkouška probíhá do okamžiku vzniku dvou nových, samostatných povrchů.

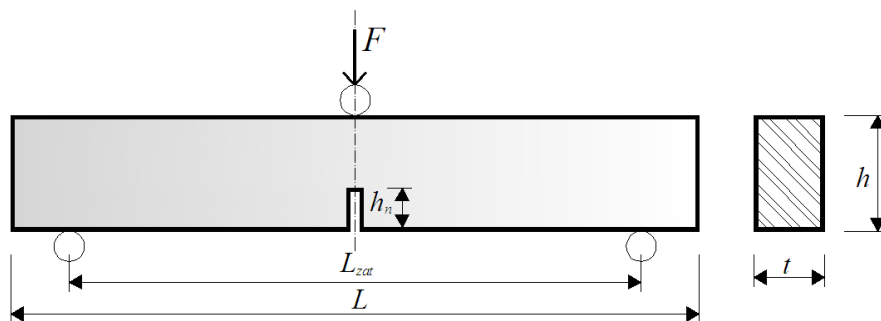
Speciálním případem zkoušky ohybem jsou zkoušky na trémecích s vruby. Vrubu různých tvarů mají vliv na formování trhliny. V místě vrubu je nejužší profil trémce, a proto





Obrázek 2.1: Diagramy ideálního vztahu napětí( $\sigma$ ) a deformace( $\epsilon$ ) pro různé typy materiálu: a) křehký, b) pružno-plastický, c) kvazikřehký, d) rozdíl mezi zpevněním a změkčením [18]

vzniká trhlina právě tam. Obrázek 2.2 zobrazuje trámec tvaru kvádru při začátku zkoušky s vrubem.



Obrázek 2.2: Schéma geometrie trámce s vrubem [19]

Jako hodnota zatížení (LOAD) je brána síla přenášející se vzorkem na podpěry. Sledovány mohou být různé deformace zkoušeného trámce. První je průhyb, tedy o kolik se v místě zatížení posunul spodní okraj trámce. Tato hodnota je velmi snadno určena u tvrdých materiálů. Avšak u pískovce dochází k zaboření podpěr do trámce. Výsledné hodnoty průhybu pak musí projít korekcí. Ta může být problematická, zvláště pokud jsou podpěry zabořeny různě. Jiným druhem deformace je rozevření ústí trhliny, označován jako CMOD (z anglického Crack Mount Opening Displacement). Tyto hodnoty lze měřit opticky, nebo nalepením extensiometru. Aby bylo zajištěno, že v místě, kde se nachází extensiometr, trhlina skutečně vznikne, musí být vzorky opatřeny zářezen – vrubem. Pak je již snadné tuto hodnotu během zkoušky měřit. Z těchto CMOD hodnot a síly působící na podpěry trámce

lze sestavit diagram, jenž je vidět na Obrázku 2.1. Pro kvazikřehký materiál je pak tento diagram lepe popsán Obrázkem 2.3.

Tvar P-CMOD diagramu je ovlivňován lomovými vlastnostmi materiálu. Právě tyto vlastnosti se snaží následná simulace ověřit, a jsou proto částí vstupu simulace:

**Lomová energie** Energie potřebná ke zpřetrhání vazeb a tvorbě dvou nových povrchů.

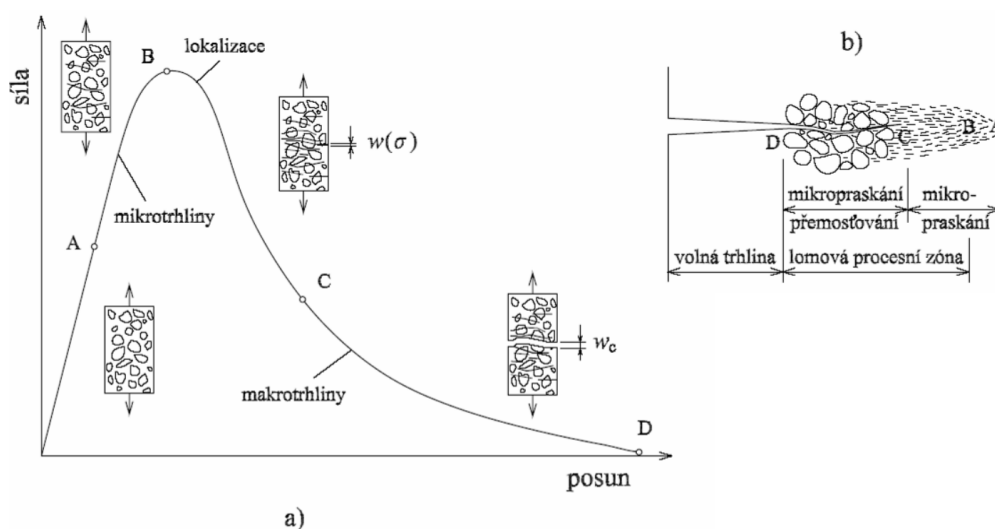
**Pevnost v tahu** Odolnost materiálu na tah. Mez pevnosti je maximální hodnota normového napětí při níž není porušena celistvost materiálu.

**Lomová houževnatost** Kritický součinitel intenzity napětí. Charakterizuje odpor materiálu ke křehkému porušení.

**Modul pružnosti** (Yonguv modul) Podíl napětí a jím vyvolané deformace.

**Poissonovo číslo** Označuje relativní prodloužení materiálu při zúžení v příčném řezu. Zúžení při namáhání tahem.

Výsledný diagram lze rozdělit na čtyři úseky pomocí čtyř bodů *A-D* zobrazených v Obrázku 2.3. Až po bod *A* je chování vzorku elastické. Stejně jako u křehkých či pružno-plastických materiálů se zde netvoří trvalé změny v materiálu, nedochází k nevratným deformacím.



Obrázek 2.3: Stav vzorku, trhliny, v různých částech P-CMOD diagramu (a) a detailní popis vznikající trhliny v materiálu (b) [12].

V další části *A-B* dochází k tvorbě mikrotrhlin uvnitř materiálu, a ten se už nemůže vrátit do svého původního stavu. V bodě *B* je vzorkem přenášena největší síla na podpěry. Od tohoto bodu po bod *C* probíhá lokalizace. Při ní se některé trhliny rozšiřují na úkor jiných. Od bodu *C* se pak dá mluvit o makrotrhlinách, které se rozšiřují až do úplného porušení materiálu v bodě *D*. V tomto posledním bodě je síla přenesená na podpěry rovna nule a trámec je rozdělen na dvě poloviny.

## Kapitola 3

# Simulace lomové zkoušky

Pro počítačové modelování různých procesů se dnes často používá metoda konečných prvků (MKP, nebo FEM z anglického finite element method). Ta rozděluje těleso na konečný počet menších částic, které dohromady tvoří abstraktní model. Přesnost výpočtu je citlivá na hustotu elementů v modelu, což klade vysoké nároky na výpočetní techniku.

Praktické pokusy se simulacemi se objevily v programu Apollo a následně v dalších specializovaných programech pro letectví či vojenství. V dnešní době se MKP díky zvýšení výkonosti počítačů využívá prakticky kdekoli.

Pro snížení časové náročnosti se dá využít zjednodušeného modelu. Některé metody pro simulaci lomové zkoušky používají například převod na 2D model. Ten se dá použít u vzorků typu kvádrů, kde je délka – hloubka vrubu konstantní. Vzorek, který je používán v této práci, je však kruhového průřezu s V vrubem vzniklým zbroušením (viz níže). Chybí zde příčná souměrnost, která by jej dovolila zjednodušit. V ose zatížení však vzorek souměrný je. Díky tomu lze simulovat pouze polovinu trámce a oblast trhliny. Druhá polovina by se měla chovat souměrně.

### 3.1 Motivace pro simulaci

Pro většinu materiálu se dlouhou dobu využívá smluvní diagram, který počítá s tolerancí pro daný materiál. Při dnešních konstrukcích se stavby pohybují daleko před bodem lomu či vzniku mikrotrhlin.

Historické budovy a stavby jsou však vystaveny vnějším vlivům a projevuje se u nich únava materiálu. Zvláště časté jsou pískovcové stavby, hrady či mosty, u kterých například vlivem eroze došlo k porušení některých stavebních bloků. Tyto bloky je potřeba vyměnit za nové. Samozřejmostí je volba stejně, nebo alespoň podobně, vypadajícího materiálu. Nový blok by měl mít co nejpodobnější únavové chování jako již stávající materiál. Například při použití pevnějšího bloku by měl časem tento blok jinou výšku než měkkí bloky kolem něj. Bloky by tak netvořily souvislou řadu. Obdobně pokud by náhrada byla měkkí než ostatní, mohlo by v tomto místě docházet k propadu řad nad tímto blokem.

Při hledání náhrady je nejdříve odebrán vzorek z budovy. Tento vzorek je nejsnáze vyvrtán, a proto má kruhový průřez. Následně je ve vzorku vytvořen vrub a aby plocha lomu zůstala co největší, je použit vrub tvaru V. Takto získaný vzorek je podroben lomové zkoušce, a tím zničen. Následně je provedena série simulací s různými předpokládanými vlastnostmi materiálu. Lze předpokládat, že závěry nelze vyvodit na základě jednoho vzorku

a celý proces se musí vykonat několikrát. S přesně určenými parametry potřebné náhrady je pak snazší určit zdroj náhradního materiálu.

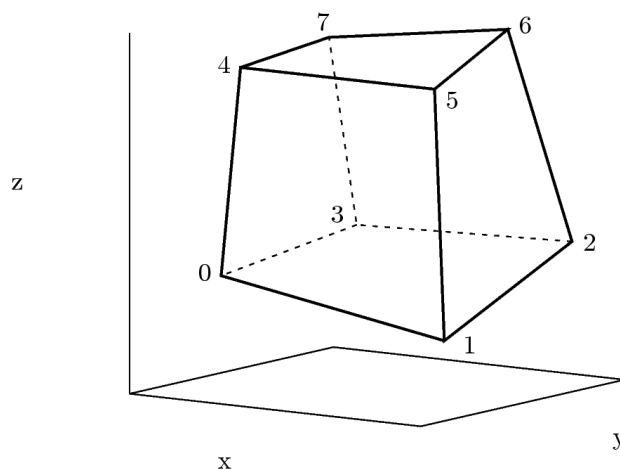
## 3.2 Vstupní model trámce

Hlavní součást vstupující do simulace je model reprezentující zkoumaný trámec. Tento model je popsán textovým souborem s následující strukturou:

- souhrn modelu,
- vazby mezi body,
- popis bodů,
- plocha pod body trhliny.

Jelikož jsou kvazikřehké materiály typicky kompozitní, je rozložení bodů i vazeb v modelu do značné části náhodné.

Body modelu dohromady tvoří primitiva, konečné elementy, které jsou ve zdrojovém kódu označovány jako *brick*. Toto označení může vyvolat představu opravdových cihel, tedy souměrných kvádrů. Ve skutečnosti, díky náhodné tvorbě bodů i vazeb, primitiva nemusí být, a z pravidla nejsou, souměrná. Příklad jednoho primitiva je na Obrázku 3.1.

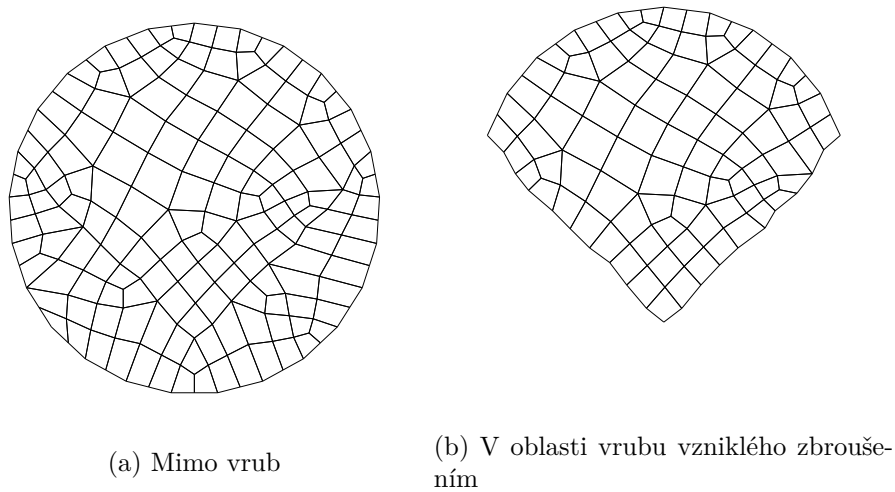


Obrázek 3.1: Příklad zobrazení jednoho primitiva *brick* v prostoru [10]

Souhrn modelu obsahuje počet jednotlivých primitiv *brick*, celkový počet bodů, počet bodů na obvodu vrubu a počet bodů vrubu na výšku. Dále je zde zaznamenáno, který bod je podepřen, zatěžován a sledován. Sledovaným bodem je míněn bod na okraji vrubu, který se při zatěžování vzdaluje ose symetrie a udává CMOD.

Následující část souboru popisuje vazby mezi jednotlivými body, tedy které body tvoří jednotlivé konečné elementy – *brick*. Každý element je popsán indexy osmi bodů. Jeden bod může být použit v různém počtu primitiv. Toho si lze všimnout na obrázku 3.2.

Samotné body jsou popsány pozicí v prostoru udanou trojicí souřadnic  $(x,y,z)$ . Čtvrtá hodnota při definici bodu pak udává jeho hmotnost. Popis bodu obsahuje také index, tedy pořadové číslo, jež koresponduje s popisem jednotlivých primitiv.



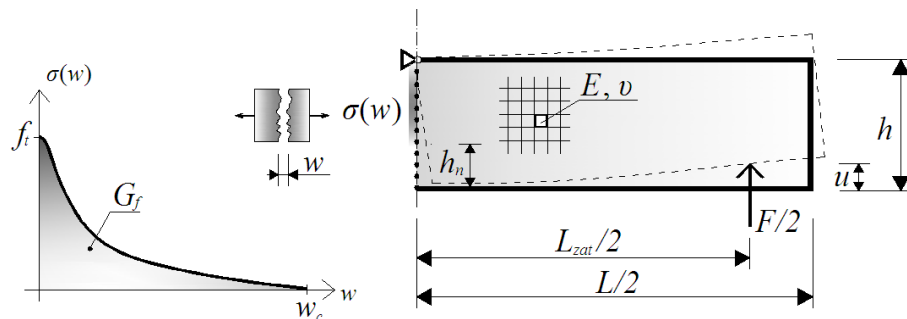
Obrázek 3.2: Uspořádání bodů v řezu modelem [10]

Pro funkci *kohezivní trhliny* jsou zde přidány i velikosti plochy pod body tvořící trhlinu. Na základě této plochy jsou určeny hodnoty mezní síly potřebné k odtržení daného bodu od osy symetrie 3.7.

Dále jsou vstupem lomové charakteristiky materiálu popsané na konci Kapitoly 2.

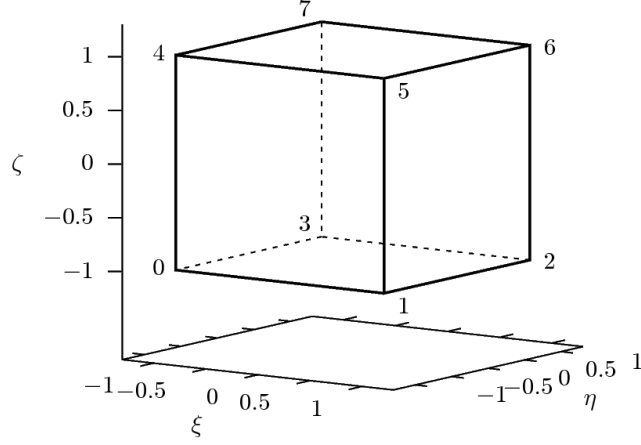
### 3.3 Příprava dat pro simulaci

Před samotnou simulací je potřeba určit mezní hodnoty síly pro odtržení jednotlivých bodů, matici tuhosti jednotlivých konečných elementů a další hodnoty, na kterých je simulace závislá. Model je vytvořen ze dvou částí. První je pružná a modeluje lineární část materiálu trávce metodou konečných prvků. Druhá je nelineární část v ose symetrie, kde je očekáván vznik kohezivní trhliny [12]. Vznik a průběh lomu zde závisí na tahové pevnosti materiálu  $f_t$ . Dokud nedojde k překročení tahové pevnosti, zůstává nelineární vrstva nad zářezem v ose symetrie. Po překročení tahové pevnosti se vrstva od osy odděluje – vzniká trhlina. Tato trhlina má dále kohezivní zónu, ve které se líce trhliny svírají kohezivním napětím  $\sigma$ . Toto napětí přitahuje oddělenou část k ose symetrie a je závislé na velikosti otevření trhliny  $w$ .



Obrázek 3.3: Model lomu: a) závislost kohezivního napětí(funkce tahového změkčení); b) parametrizace modelu [19]

Funkční závislost  $\sigma(w)$  je rovněž nazvána jako tahové změkčení a definuje se tahovou pevností  $f_t$  a lomovou energií  $G_f$ , která odpovídá ploše pod křivkou. Výpočtem lze stanovit kritické rozevření trhliny  $w_c$ , při němž přestává působit kohezivní napětí a trhlina je považována za volnou.



Obrázek 3.4: Příklad zobrazení jednoho primitiva *brick* po transformaci do souřadného systému  $\xi\eta\zeta$ . [10]

Nejdůležitější částí simulace je matice tuhosti  $[K]$ , vypočtená pro všechny prvky modelu. Každý bod je reprezentován rychlostí, pozicí v trojrozměrném prostoru a hmotností. Aby bylo možné stanovit tuhost jednotlivých prvků modelu, je potřeba transformovat vrcholy prvků. Ze souřadného systému  $(x,y,z)$ , který znázorňuje Obrázek 3.1, do souřadného systému  $(\xi\eta\zeta)$ , na Obrázku 3.4. V tomto systému může pozice bodu nabývat pouze hodnoty 1 nebo -1.

Na základě transformovaných souřadnic jsou určeny *tvarové funkce*  $N$  pro všech osm vrcholů. Rovnice 3.1 odpovídá rovnici  $N$  pro vrchol 0. Ostatní rovnice  $N$  vzniknou změnou znamének u parametrů  $\xi$ ,  $\eta$  a  $\zeta$ .

$$N_0 = \left| \frac{1}{8} * (\xi - 1) * (\eta - 1) * (\zeta - 1) \right| \quad (3.1)$$

Podle těchto tvarových funkcí je vytvořena matice lokálních derivací  $[\partial L]$ , a ta je následně vynásobena s maticí původních vrcholů elementu  $[p]$ . Vzniklý produkt je Jakobiho matice  $[J]$ .

$$[J] = [\partial L] * [p] = \begin{bmatrix} \frac{\partial N_0}{\partial \xi} & \frac{\partial N_1}{\partial \xi} & \frac{\partial N_2}{\partial \xi} & \frac{\partial N_3}{\partial \xi} & \frac{\partial N_4}{\partial \xi} & \frac{\partial N_5}{\partial \xi} & \frac{\partial N_6}{\partial \xi} & \frac{\partial N_7}{\partial \xi} \\ \frac{\partial N_0}{\partial \eta} & \frac{\partial N_1}{\partial \eta} & \frac{\partial N_2}{\partial \eta} & \frac{\partial N_3}{\partial \eta} & \frac{\partial N_4}{\partial \eta} & \frac{\partial N_5}{\partial \eta} & \frac{\partial N_6}{\partial \eta} & \frac{\partial N_7}{\partial \eta} \\ \frac{\partial N_0}{\partial \zeta} & \frac{\partial N_1}{\partial \zeta} & \frac{\partial N_2}{\partial \zeta} & \frac{\partial N_3}{\partial \zeta} & \frac{\partial N_4}{\partial \zeta} & \frac{\partial N_5}{\partial \zeta} & \frac{\partial N_6}{\partial \zeta} & \frac{\partial N_7}{\partial \zeta} \end{bmatrix} * \begin{bmatrix} x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \\ x_5 & y_5 & z_5 \\ x_6 & y_6 & z_6 \\ x_7 & y_7 & z_7 \end{bmatrix} \quad (3.2)$$

Dalším krokem je sestavení matice stress-strain  $[B]$  materiálu. Ta je určena z globální derivace  $[\partial G]$ , jejíž členy jsou vloženy na korespondující místa matice  $B$ .

$$[\partial G] = [J^{-1}] [\partial L] \quad (3.3)$$

$$[B]^T = \begin{bmatrix} \partial G_{11} & 0 & 0 & \partial G_{21} & 0 & \partial G_{31} \\ 0 & \partial G_{21} & 0 & \partial G_{11} & \partial G_{31} & 0 \\ 0 & 0 & \partial G_{31} & 0 & \partial G_{21} & \partial G_{11} \\ \partial G_{12} & 0 & 0 & \partial G_{22} & 0 & \partial G_{32} \\ 0 & \partial G_{22} & 0 & \partial G_{12} & \partial G_{32} & 0 \\ 0 & 0 & \partial G_{32} & 0 & \partial G_{22} & \partial G_{12} \\ \partial G_{13} & 0 & 0 & \partial G_{23} & 0 & \partial G_{33} \\ 0 & \partial G_{23} & 0 & \partial G_{13} & \partial G_{33} & 0 \\ 0 & 0 & \partial G_{33} & 0 & \partial G_{23} & \partial G_{13} \\ \partial G_{14} & 0 & 0 & \partial G_{24} & 0 & \partial G_{34} \\ 0 & \partial G_{24} & 0 & \partial G_{14} & \partial G_{34} & 0 \\ 0 & 0 & \partial G_{34} & 0 & \partial G_{24} & \partial G_{14} \\ \partial G_{15} & 0 & 0 & \partial G_{25} & 0 & \partial G_{35} \\ 0 & \partial G_{25} & 0 & \partial G_{15} & \partial G_{35} & 0 \\ 0 & 0 & \partial G_{35} & 0 & \partial G_{25} & \partial G_{15} \\ \partial G_{16} & 0 & 0 & \partial G_{26} & 0 & \partial G_{36} \\ 0 & \partial G_{26} & 0 & \partial G_{16} & \partial G_{36} & 0 \\ 0 & 0 & \partial G_{36} & 0 & \partial G_{26} & \partial G_{16} \\ \partial G_{17} & 0 & 0 & \partial G_{27} & 0 & \partial G_{37} \\ 0 & \partial G_{27} & 0 & \partial G_{17} & \partial G_{37} & 0 \\ 0 & 0 & \partial G_{37} & 0 & \partial G_{27} & \partial G_{17} \\ \partial G_{18} & 0 & 0 & \partial G_{28} & 0 & \partial G_{38} \\ 0 & \partial G_{28} & 0 & \partial G_{18} & \partial G_{38} & 0 \\ 0 & 0 & \partial G_{38} & 0 & \partial G_{28} & \partial G_{18} \end{bmatrix}^T \quad (3.4)$$

Výsledná matice tuhosti  $[K]$  pro daný element je pak spočtena podle Vzorce 3.5.

$$[\partial K] = \sum_i 1^8 |det(J)| * B_i^T * D * B_i * h_i \quad (3.5)$$

V této rovnici  $[J]$  je Jacobiho matice pro daný element,  $det(J)$  je determinant této matice,  $B_i$  je část matice  $[B]$  pro bod  $i$  a  $h_i$  je váhová funkce [10].

### 3.4 Běh simulace

Při běhu simulace je v každém kroku posunut jeden bod modelu, bod zatížení, ve směru tohoto zatížení. Pro všechny elementy je následně spočtena hodnota vnitřních sil, reakcí  $R$ ,

působících na body elementu. Ta je spočtena jako rozdíl pozic oproti původním přechtělým ze vstupního modelu. Matice tuhosti pak určí, jak moc ovlivňuje daný posun pozice všech bodů *brick*.

Následně jsou *Eulerovou metodou* aktualizovány rychlosti a podle nich i pozice bodu. Pro výpočet aktuální rychlosti částice  $v(t+h)$  je kromě reakce  $R$ , hmotnosti  $m$  a předešlé rychlosti  $v(t)$  použito také viskozitní tlumení  $c$ . Pro aktuální pozici  $p(t+h)$  je pak použita pozice předchozí  $p(t)$  a aktuální rychlost  $v(t+h)$ . Dané rovnice pak vypadají následovně:

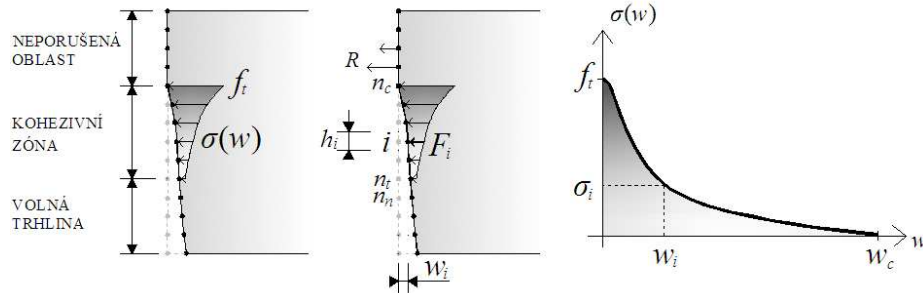
$$\begin{aligned} v_x(t+h) &= v_x(t) + h \left( \frac{R_x - cv_x(t)}{m} \right) \\ v_y(t+h) &= v_y(t) + h \left( \frac{R_y - cv_y(t)}{m} \right) \\ v_z(t+h) &= v_z(t) + h \left( \frac{R_z - cv_z(t)}{m} \right) \end{aligned} \quad (3.6)$$

$$x(t+h) = x(t) + hv_x(t+h)$$

$$y(t+h) = y(t) + hv_y(t+h)$$

$$z(t+h) = z(t) + hv_z(t+h)$$

### 3.5 Algoritmus pro výpočet formování trhliny



Obrázek 3.5: Formování trhliny v závislosti na funkci kohezivního napětí [19]

Každému uzlu  $i$  realizujícímu připojení trámce v ose symetrie je zamezen posun ve směru osy  $x$  do okamžiku, než jeho reakce  $R_i$  nepřekročí kritickou hodnotu  $F_{c,i}$  danou jako:

$$F_{c,i} = f_t \cdot A_i \quad (3.7)$$

kde  $f_t$  je pevnosti v tahu materiálu dána funkcí kohezivního napětí a  $A_i$  představující plochu části průřezu přiřazené danému uzlu  $i$ . Pokud došlo k porušení pružiny a reakce je větší než  $F_{c,i}$ , je uzel  $i$  uvolněn v ose  $x$  a působí na něj kohezivní síla  $F_i$ :

$$F_i = A_i \cdot \sigma(w_i) \quad (3.8)$$

Při překročení posunu  $w_i$  nad mez  $w_c$  je kohezivní napětí  $\sigma$  rovno nule a kohezivní síla již na bod nepůsobí. V obrázku 3.5 jsou proto zobrazeny případy *volná trhlina*, kdy na body působí pouze síla lineární části modelu, *kohezivní zóna*, kde na body působí, jak kohezivní síla, tak síla lineární části a *neporušená oblast*, kde na body v ose symetrie působí opět pouze síla lineární části, která doposud nepřesáhla meze odtržení.



## Kapitola 4

# Analýza původní simulace

Tato kapitola popisuje, jak je metodika simulace převedena do zdrojového kódu a které části byly určeny jako slabá místa. Kapitola 6 následně popíše, jak byly tato problematická místa simulace optimalizována.

### 4.1 Popis implementace

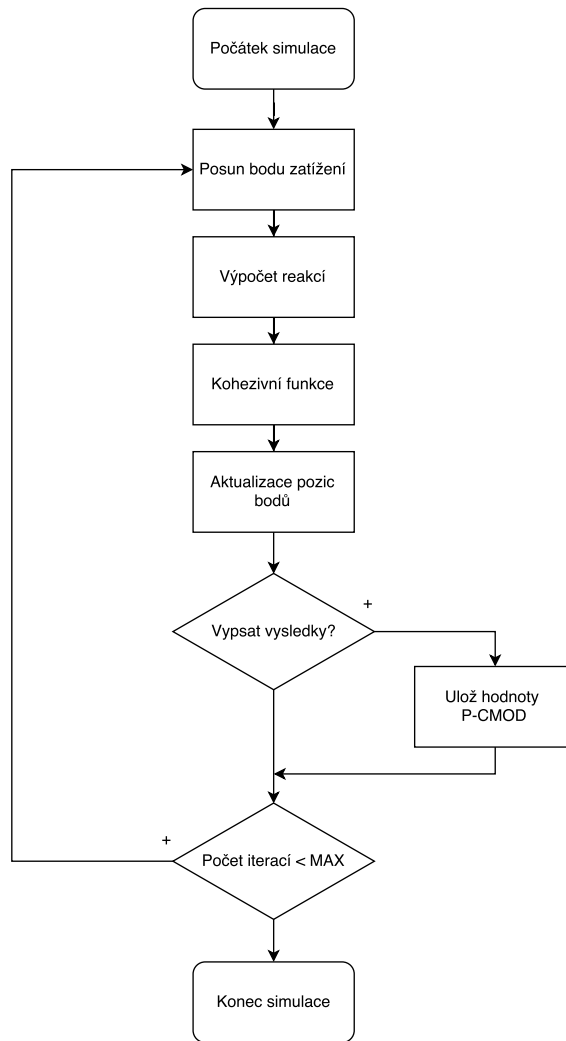
Původní kód byl sepsán v jazyce C/C++ a lze rozdělit na dvě části. První čte vstupní soubor popsáný v Kapitole 3.2 a na základě něj a zadaných parametrů sestavuje matici tuhosti důležitou pro další část. Tou je samotná simulace, která každý krok posouvá konstantně jeden bod a určuje, jak na tento posun ostatní body zareagují.

Načtený soubor je zpracováván po řádcích. První řádek obsahuje počet elementů a bodů v souboru, stejně jako informace o počtu bodů tvořících trhlínu. Díky tomu lze hned na počátku alokovat dostatečné množství paměti pro všechna načtená data. Současně je alokována i paměť pro ukládání mezivýpočtů potřebná pro koordináty všech bodů, jejich rychlosti, reakce a již zmíněnou matici tuhosti. Po přípravě paměti jsou data načtena. Dle struktury souboru se nahrávají nejdříve data pro elementy *brick*, následně body a na konec plocha pod body trhlíny. Všechna načtená data jsou čísla, a to buď hodnoty jako hmotnost či posunutí od počátku v 3D prostoru, nebo indexy odkazující na jednotlivé body. O převod z řetězce na číslo se starají funkce *atoi()* a *atof()*.

Posledním krokem inicializační části je sestavení matic tuhosti pro všechny konečné elementy *brick* podle rovnic popsáných v Kapitole 3.5. Pozice bodu jsou uloženy ve třech polích, kde každé obsahuje pouze jednu ze tří souřadnic  $x, y$  a  $z$ . To zajišťuje efektivní práci procesoru s těmito souřadnicemi. Jednotlivé body jsou uspořádány v pořadí odpovídajícím jejich definici v souboru. Tato inicializace je konstantní prací vykonávanou pro daný model a dala by se označit za  $\alpha$  dle Amdahlova zákona. Díky vysokému počtu iterací simulační části se jedná o zanedbatelný zlomek výpočtu.

Následující část obstarává samotnou simulaci. Je řízena parametrem *step*, neboli krokem určujícím časovou vzdálenost mezi jednotlivými iteracemi a samotným počtem těchto iterací. Je potřeba, aby byl krok co nejmenší, řádově  $10^{-6}$  sekund a menší, jinak je metoda nestabilní a často jsou výsledkem hodnoty NAN (Not A Number). Během každé iterace je proveden výpočet působících reakcí uvnitř prvků *brick* a na základě těchto reakcí je určen pohyb bodů.

Výpočet reakcí uvnitř každé *brick* závisí na změně pozic jednotlivých bodů od jejich výchozích pozic a na matici tuhosti. Pro každý *brick* je tato matice diagonálně symetrická



Obrázek 4.1: Schema simulační částí programu.

a má rozměry  $24 \times 24$ . Jedna buňka matice udává, jak pohyb na pozici  $i$  ovlivňuje  $j$ . Nejdříve je určena změna v pozicích všech osmi bodů. Jelikož je každý bod určen vektorem hodnot  $x, y, z$ , je pro element obdrženo 24 hodnot označovaných jako *deformace*. Tato deformace je pak opakovaně procházena pro všechny souřadnice *brick* a přičtena jako jeho reakce. Jeden bod je většinou součástí více elementů.

Reakce působící na body trhliny je potřeba upravit pomocí kohezivní funkce. Ta určuje množství reakce působící na body druhou půlkou trámce, která není modelovaná.

Po vypočtení těchto reakcí jsou aktualizovány pozice jednotlivých bodů. Reakce jsou opět uloženy zvlášť pro každou souřadnici. Jedno pole obsahuje reakce pro souřadnice na ose  $X$  a další dvě na zbývajících. Výchozí rychlost všech bodů je nulová. Během iterací se na základě hmotnosti bodu, vypočtených reakcí a koeficientu tlumení mění. Nejdříve je vždy určeno zrychlení v dané iteraci. Eulerovou metodou je pak podle kroku simulace a tohoto zrychlení určena rychlost bodu v daném kroku a jeho následující pozice.

## 4.2 Využití procesoru

PAPI [8] (Performance API) je rozhraní umožňující přistupovat specializovaným čítačům procesoru pro měření výkonu. Výsledky měření některých výkonových charakteristik jsou zobrazeny v Tabulce 4.1.

Měření bylo provedeno na jednovláknové aplikaci a na aplikaci rozdělující *bricks* mezi osm jader. Původní kód neobsahoval schopnost tohoto rozdělení na vlákna. Ta byla přidána před provedením testování. Hodnoty s danou jednotkou, Flops, jsou ve sloupci "8 jader" sečteny. Ostatní hodnoty udávány v procentech jsou v posledním sloupci průměrovány.

Jedním z charakteristických parametrů programu je jeho výpočetní rychlost. Ta se udává v počtu zpracovaných operací v řádové čárce za sekundu označované právě jako Flops, případně řádově vyšších MFlops či GFlops. V tabulce si lze všimnout, že původní výkon v sekvenční verzi je 0,9 GFlops. Testovaný procesor má teoretický maximální výkon 19,2 GFlops<sup>1</sup>. Původní implementace tedy využívá pouze 4,68 procent potenciálu procesoru. Při paralelizaci na osm jader však není výkon 8 krát větší, ale pouze 5,57 krát. Se ztrátou výkonu na jádro se dá počítat vlivem režie paralelizace.

Paměti cache L2 jsou vázány na jádro. Paměť L3 je pak mezi vlákny sdílená a znatelně větší. Konkrétně u použitého referenčního procesoru se jedná o L2 cache o velikosti 256 KB na jádro a L3 cache 20 MB pro všech osm jader. Výpadky v L2 cache pamětech znamenají, že hodnoty jsou hledány v L3 cache. Obě tyto paměti jsou poměrně rychlé. Výpadky v L3 znamenají, že hodnoty jsou načítány z hlavní paměti a toto načítání je značně pomalejší.

Při práci jednoho jádra má toto jádro svou L2 cache. Zároveň má pro sebe i výrazně větší L3 cache, kterou by standardně sdílelo s ostatními jádry. Jelikož ostatní jádra nepracují, může být tato L3 cache celá využita jedním jádrem a značná část úlohy je načtena do této cache. Techniky procesoru pro přednačítání do cache pak zajišťují, že výpadky v L3 jsou minimální.

Stejně tak počet cyklů procesoru, kdy procesor na něco čeká, jsou u jednojádrového provedení velmi nízké. Typickým důvodem pro čekání bývá právě nepřítomnost dat v cache a potřeba obsloužit načtení jiného bloku dat. Procesor neustále vykonává nějaké instrukce, netráví čas čekáním. Nízký výkon ve Flops naznačuje, že procesor pravděpodobně používá k výpočtu jen pár funkčních jednotek, zatímco ostatní jsou nevyužity. Nebo vykonává řídicí instrukce, kterými jsou například podmíněné skoky.

Výpadky cache u paralelní verze vzrostly. Nárůst v L2 o přibližně dvě procenta způsobují techniky pro koherenci paměti cache. Tyto techniky se starají o zneplatnění dat v L1 a L2 cache jader, pokud jedno jádro změní hodnotu, kterou mají nahránu jiné. Skutečnost, že se jádra musí dělit o L3, cache vede k zvýšení počtu výpadků v L3 takřka o dva řády. Spolu s tím musel zákonitě jít nahoru i počet cyklů, kdy procesor nevystavil žádnou instrukci, a tedy čekal na potřebný blok dat. Ten vzorstl desetinásobně.

Další charakteristikou simulačního nástroje může být počet elementů nebo iterací na jednotku času, případně doba na element či iteraci. Tyto hodnoty jsou obsaženy v Tabulce 4.2. Je vhodné podotknout, že model, na němž bylo prováděno měření, obsahuje pouze 2310 *brick* elementu. Tedy hodnota v tabulce udávající počet zpracovaných elementů za sekundu je potencionální rychlost při dostatečně velkém modelu. Třetí sloupec tabulky udává výkon jednoho jádra jako 1/8 výkonu paralelizované verze. Tento výkon je nižší než u sekvenční verze kvůli zmíněným výpadkům v L3 cache.

<sup>1</sup><https://docs.it4i.cz/anselm/compute-nodes/>

Tabulka 4.1: Tabulka výkonových parametrů původní implementace

	Jedno jádro	Medián z osmi jader	Osm jader
MFLOPS	936,36	794,82	6 353,22
Vypadky v L2 cache	6,88 %	8,6 %	8,9 %
Výpadky v L3 cache	0,0992 %	8,845 %	8,837 %
Cykly bez instrukce	0,0225 %	0,234 %	0,289 %

Tabulka 4.2: Tabulka rychlosti zpracování iterací původní implementací

	Sekvenčně Jedno jádro	Na osmi jádrech Celkem	Na jádro
Čas na iteraci	2,782ms	0,461ms	3,688ms
Iterací za sekundu	359	2 168	271
Čas na element	1,204 $\mu$ s	0,199 $\mu$ s	1,596 $\mu$ s
Elementu za sekundu	830 564	5 009 759	626 219

### 4.3 Slabé místo

Dle původní analýzy programem Alinea Map [1], více než 60 procent času výpočtu připadá na jeden řádek C++ kódu. Tento řádek je nejvnitřnějším tělem několika *for* smyček simulace. Vnější smyčka *for* je určena pro posun simulace v čase. Uvnitř leží výpočet reakcí v modelu, aktualizace bodů konečných elementů a aktualizace kohezivní vazby trhliny. Nejslabším místem je úsek, který prochází deformace a matice tuhosti.

Rozbor tohoto řádku programem Alinea Map je možné vidět jako výřez obrazovky na Obrázku 4.2a.

Ten říká, že skalárním, po jedné operaci zpracovávajícím, výpočtem hodnoty v plovoucí řádové čárce zde tráví procesor 78,9 procent času. Zbytek připadá čekání na paměť.

Pro každý *brick* v každé iteraci je kritické místo navštíveno 24\*24 tedy 576 krát. Tento úsek kódu je tedy z pohledu optimalizace nejzajímavější. Z pohledu složitosti můžeme říct že kód má v tomto useku složitost  $O(n) = 576 * n$  kde  $n$  je počet *brick*.

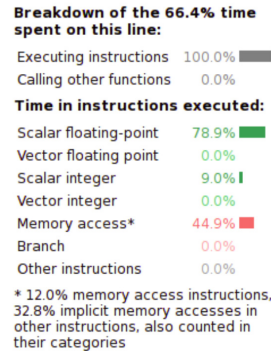
Oproti tomuto místu mají zbývající procesy uvnitř jedné iterace výrazně nižší složitost. Aktualizace pozic bodu i vazeb kohezivní trhliny jsou ukoly se složitostí blíží se  $O(n) = n$ . Každý bod i vazba jsou procházeny pouze jednou iterací a na základě předchozích výpočtů jsou upraveny.

### 4.4 Vliv překladače

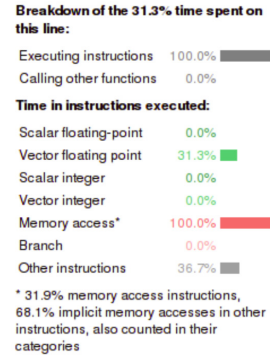
GNU kompilátor [2] je běžně používán pro překlad zdrojových kódů. Jednou z jeho komerčních alternativ je kompilátor vyvíjený firmou Intel [4]. Intel kompilátor (ICC), vyvíjen a zašitován formou Intel, má určitý technologický náskok oproti GNU kompilátoru (GCC) od neziskové organizace Free Software Foundation. Intel kompilátor tak poskytuje dříve integraci nových technologií procesoru a jejich lepší podporu. Jedním z příkladů může být vektorizace. Na obrázku 4.2 jsou zobrazeny výstupy běhu simulace stejných vstupních dat.

Byla použita GCC verze 4.9.0 a 16.0.1 ICC.

- Velikost modelu 1370 *brick*.
- Běh programu kompilovaném GCC byl 2 699,8s (45 minut).
- Běh programu kompilovaném ICC byl 1 365,8s (23 minut).
- Dosažené zrychlení 1,976.



(a) GNU překladač



(b) Překladač firmy Intel

Obrázek 4.2: Analýza kritické sekce simulace programem Alinea Map.

Oba případy měly povolenou vektorizaci. GCC vyhodnotil, že vektorizace daného místa není výhodná. Ani po označení pomocí speciálních pragmat pro překladač, nedošlo pod GCC k vektorizaci. Z důvodu lepší dostupnosti GCC, zvláště v porovnání s cenou ICC, jsou další podkapitoly zaměřeny právě na GCC, přesněji ve verzi 4.9.0. Mělo by být možné dosáhnout podobných výsledků jako u ICC, zlepšení běhu na GCC by taktéž mělo urychlit kód kompilován pomocí ICC.

## 4.5 I/O

Dalším z potenciálních slabých míst jsou I/O operace. Přístupová doba k RAM je podstatně nižší než přístupová doba k disku<sup>2</sup>. Kdežto propustnost disku se pohybuje kolem 10MB/s, propustnost RAM disku se pohybuje kolem 5-50GB/s. Ne všechny výpočetní sestavy však mají mapovaný souborový systém typu RAM disk. Uvnitř hlavní smyčky je taktéž průběžný tisk výsledků. Ten je prováděn pomocí funkce `fprintf()` a následného `fflush()`. Výsledných dat přitom není v rámci iterace, při které se tiskne, mnoho. Jedná se pouze o dvojici bodů ležící na křivce P-CMOD diagramu. Tento zápis, přesněji požadavek na vyprázdnění bufferu pomocí `fflush()`, může prodlužovat čas simulace.

Bez vyprázdnění bufferu nejsou k dispozici žádné výsledky, dokud probíhá výpočet. Navíc, chybí i informace o tom, v jaké fázi výpočtu se simulace nachází. Pokud se výpočet dostane z nějakého důvodu do neočekávaného stavu a například začne cyklit, uživatel se o této skutečnosti nedozví. To může být problémem zvláště u dlouhých výpočtů, které probíhají hodiny až dny.

<sup>2</sup><https://docs.it4i.cz/anselm/storage/>

## Kapitola 5

# Metody optimalizace

Pravdivost Moorova zákona, tedy toho, že se co 18 měsíců zdvojnásobí počet tranzistorů na ploše čipu při zachování stejné ceny, zajišťuje v poslední době paralelizace [17]. Ta je realizovaná na více úrovních. Pomocí vektorizace lze uplatnit přístup Single Instruction Multiple Data (SIMD). Při tomto přístupu je procesor, přesněji jádro tohoto procesoru, schopno v jednom taktu zpracovat jednu operaci více dat. Zvyšování počtu jader procesoru na jednom čipu a propojování procesorů do výpočetních klastrů umožňuje přístup *multiple instruction multiple data* MIMD, často programovaný podle vzoru SPMD *Single Program Multiple Data*. V tomto případě si jednotlivá jádra procesoru mohou rozdělit zpracováváný program. Buď tak, že každé jádro vykonává část programu, nebo si jádra rozdělí data a každé zpracovává svou část. Dalším přístupem je výpočet na grafické kartě s využitím CUDA. Moderní grafické karty jsou složeny z několika SIMD procesorů a tyto procesory mohou pracovat společně nebo nezávisle. Všechny technologie lze kombinovat. Lze využít skupinu procesů na jednotlivých výpočetních strojích komunikujících pomocí MPI [6]. Na každém stroji lze rozdělit práci do skupin OpenMP [7] vláken využívajících SIMD a zároveň provádět část výpočtu na grafické kartě. Otázkou však je, zda režie pro takovou distribuci není větší, než kdyby byl využit třeba jen jeden přístup.

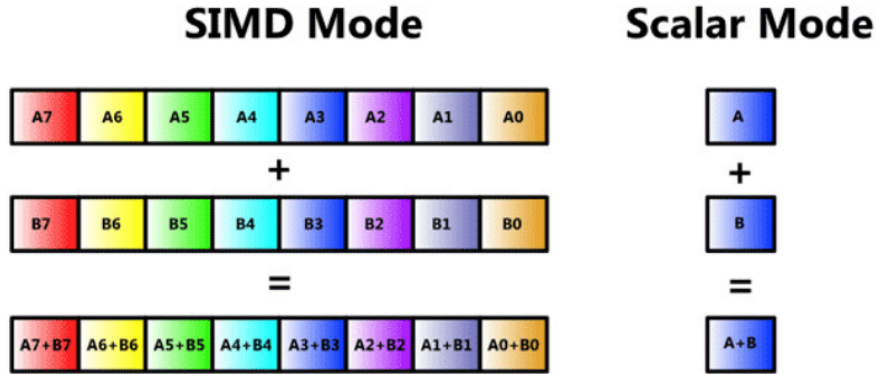
### 5.1 Vektorizace - SIMD

Single Instruction Multiple Data si lze představit jako několik stejně výkonných jednotek procesoru, které přijímají na vstupu rozdílná data a stejnou instrukci. Může se jednat o sčítání, násobení a další. Vykonání instrukce probíhá současně a na výstupu jsou opět obecně různá data. Pro SIMD operace musí však být procesor vybaven adekvátním hardware. Běžně používaný systém je SIMD Within A Register (SWAR), kde je procesor doplněn o speciální registry a výpočetní jednotky. V takovém případě je potřeba tyto specializované registry před výpočtem naplnit a po výpočtu výsledky opět rozeslat, uložit, na příslušné místo v paměti.

SIMD se v běžných procesorech objevuje v první řadě u procesorů firmy Intel. Ta roku 1997 představila rozšíření MMX. To disponovalo osmi 64 bitovými registry pro celá čísla. Do těchto registrů se mohla nahrát dvě 32bitová čísla, čtyři 16bitová či osm 8bitových. Následně mohl procesor vykonávat nad těmito registry takzvané zabalené (packed) instrukce, které zpracovaly daný počet čísel najednou.

Významější je však rozšíření procesoru SSE z roku 1999, které bylo dostupné v procesorech Pentium III. Toto rozšíření do procesoru přidává osm 128bitových registrů, které

můžou být použity pro čtyři čísla v plovoucí řádové čárce (float). Jeho následník SSE-2 dokázal tyto registry využít i pro jiné datové typy.



Obrázek 5.1: Rozdíl mezi skalární a vektorovou (SIMD) operaci. [15]

V podobném duchu se neslo vylepšování této technologie i v dalších letech. Některá rozšíření spočívala ve změně šířky registrů, jiná zvýšila počet datových typů, které tyto registry mohou zpracovávat. AVX zvýšil délku registru na 256bitu s možností využít osm *float* hodnot v jednom registru nebo čtyřmi *double*. AVX2 již uměl využít registry i pro celá čísla různých délek. Rozšíření AVX-512 představené roku 2015 opět rozšiřuje registry na délku 512bitu. Každé rozšíření je zpětně kompatibilní s předchozím, a to tím způsobem, že SSE instrukce na AVX registrech pracují jen s první půlkou registru a zbytek není využíván.

Vektorizace se nevyplatí, pokud je potřeba zpracovat několik málo elementů. Je potřeba, aby načtení a rozeslání dat mohlo být překryto výpočtem. Toho je možno docílit dvěma cestami, které je vhodné kombinovat. První je zajištění dostatečné výpočetní intenzity. Tedy vysoký počet instrukcí je proveden nad jedněmi vstupními daty. Tím vzroste doba výpočtu nad jedněmi daty a procesor má možnost předpřipravit data na další výpočet. Druhým je zlevnění ceny načtení a rozeslání dat. Toho lze dosáhnout zarovnáním dat v paměti a správnou strukturou těchto dat. Data načítaná do jednoho vektorizačního registru by tak měla být v paměti řazena zasebou. Pokud by byla rozptýlena, je potřeba načíst více bloků dat, a tím je operace přípravy dat dražší.

## 5.2 Zákony o paralelizaci

Amdahlův zákon říká, že existuje část programu, která nelze paralelizovat a je potřeba jí vykonávat sekvenčně [9]. Pro aplikaci tohoto zákona je dobré ujasnit si dva pojmy. Účinnost či efektivita paralelizace  $E$  a zrychlení  $S$ . Na Rovnici 5.1 je zobrazen výpočet zrychlení, který se dá určit jako poměr času potřebného k sekvenčnímu zpracování  $T_S$  a času potřebného k paralelnímu zpracování  $T_P$ .

$$S = \frac{T_S}{T_P} \quad (5.1)$$

Efektivita  $E$  zpracování se pak dá určit jako poměr dosaženého zrychlení  $S$  k počtu použitých procesních jednotek  $P$

$$E = \frac{S}{P} \quad (5.2)$$



Zde platí, že zrychlení  $S$  by mělo vždy být menší nebo rovno počtu použitých procesních jednotek, neboli  $E$  by nemělo být větší jedna. Pokud taková situace nastává, mluvíme o takzvaném superlineárním zrychlení. K němu může dojít, například pokud pro získání času  $T_S$  nebyl použit neoptimálnější algoritmus dostupný pro sekvenční verzi.

Část programu, kterou nelze paralelizovat, je označována jako  $\alpha$ . Tato část je vykonávána vždy původní rychlostí. Zbývající práce  $(1-\alpha)$  je část programu, která lze paralelizovat. Aby Amdahlův zákon platil, je předpokládáno, že práce je konstantní a nezávislá na počtu procesních jednotek. Čas paralelní verze můžeme vyjádřit vztahem 5.3

$$T_P = \alpha * T_S + (1 - \alpha) * \frac{T_S}{P} \quad (5.3)$$

Pokud by tento vztah byl dosazen do rovnice pro zrychlení (5.1), dala by se určit limita tohoto paralelního zrychlení pro rostoucí počet procesních elementů. Tou je  $1/\alpha$ . Pro efektivitu pak platí, že limita s rostoucím počtem procesních elementů klesá k nule. To napovídá, že úlohy s konstantní sekvenční částí nelze efektivně rozdělit mezi vysoký počet procesních elementů. Z tohoto zákona taktéž plyne princip silné škalovatelnosti, tedy jak je výpočet rychlejší při stejně velké uloze na zvětšujícím se počtu procesorů.

Dalším zákonem paralelizace je Gustavsonův zákon [14]. Ten je odvozen z Amdahlůva zákona. Stanovuje, že s rostoucí velikostí úlohy  $\alpha$  relativně klesá. Tato formulace uvažuje za konstantní čas  $T_P$ . Paralelizovatelná část lineárně roste s počtem elementů a sekvenční čas by se dal vyjádřit pomocí vztahu 5.4.

$$T_S = \alpha_G * T_P + (1 - \alpha_G) * T_P * P \quad (5.4)$$

Efektivitu dle Gustavsonova zákona lze pak vyjádřit vztahem 5.5. Z něj je patrné, že jeho limita jde k  $(1 - \alpha_G)$ , a tedy škáluje lépe než efektivita u Amdahlůva zákona. Rozdíl je právě ve velikosti práce, kterou je potřeba vykonat. U Amdahlůva zákona je tato práce konstantní, u Gustavsonova roste lineárně s  $P$ . Hovoříme zde tedy o slabé škalovatelnosti.

$$E = 1 - \alpha_G + \frac{\alpha_G}{P} \quad (5.5)$$

Dosazením do rovnic pro určení zrychlení lze určit vztah mezi  $\alpha$  a  $\alpha_G$ .

$$\alpha_G = \frac{\alpha * P}{1 + \alpha(P - 1)} \quad (5.6)$$

### 5.3 Vláknoový paralelizmus

V mikroarchitektuře procesoru bylo dosaženo značného zrychlení přidáním pokročilých technologií pro zpracování jednoho programu, neboli vlákna. Mezi tyto technologie patří:

- L1-L3 cache,
- zpracování instrukcí mimo pořadí,
- predikce skoků,
- odstranění konfliktů, závislosti mezi instrukcemi,
- zvyšování frekvence procesoru,



- superskalární zpracování.

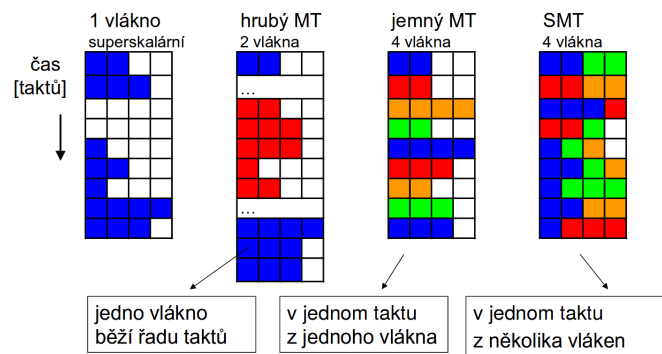
Pokusy pokračovat v rozvoji těchto technik již nevedou k prokazatelnému nárůstu výkonu. Brání tomu například paměťové latence. Využití funkčních jednotek se pohybuje kolem 30 procent. Jedním z možných řešení jak dále zvyšovat výkon procesoru je multithreading.

Multithreading využívá skutečnosti, že jedno vlákno obsahuje místa, kde nepracuje, čeká na data, nějakou událost, nebo využívá jen část dostupných prostředků. Samotná vlákna pracují ve stejném adresním prostoru, jsou lehčí a mají menší režii než procesy. To umožňuje vlákna efektivněji přepnout. Samotné vlákno lze představit jako proud instrukcí. Existují tři druhy přepínání vláken.

- 1) **Časový multithreading** (TMP -Temporal MT) je případ, kdy se vlákna na jednom výpočetním jádře střídají časově, za účelem vykrytí výpadku v cache. Zatímco jedno vlákno čeká na data, druhé zpracovává ta svá.
- 2) **Prostorový multithreading** Vlákna běží paralelně na více jádrovém procesoru. Každé vlákno má své jádro procesoru.
- 3) **Čipový multiprocessing/multithreading** Kombinuje obě předchozí a dovoluje více vláknům běžet na jednom jádru v systému s několika jádry.

Ve všech případech je potřeba mít pro přepnutí vlákna co nejkratší dobu. Každé vlákno má svůj hardware kontext, který se skládá z programového čítače, TLB a dalších důležitých registrů procesoru, které jsou v procesorech s podporou multithreadingu několikrát replikovány.

Jako další klasifikace multithreadingu se používá informace o tom, kdy se přepínají jednotlivá vlákna.



Obrázek 5.2: Příklad fungování multithreadingu podle přepínání vláken. Jednotlivé čtverce představují funkční jednotky. Ty jsou využité, každé vlákno má svou barvu, nebo nevyužité, čtverec je bílý.

- 1) **Hrubý multithreading** Přepnutí nastává jako reakce na nějakou událost, například výpadek v L2 cache. Je vhodný pro dlouhé doby blokování.
- 2) **Jemný multithreading** Vlákna se přepínají cyklicky v každém taktu. Ta která čekají jsou přeskočena. Zvládá dlouhé i krátké výpadky. Výhodou je nulová doba přepnutí kontextu.

**3) Simultální MT** V jednom taktu jsou vybaveny instrukce z více vláken. Kontext je přepnut okamžitě a větší počet vláken se cyklicky střídá.

Technologie Hyper-Threading od firmy Intel je implementací Simultálního multithreadingu. Tato funkce je u procesorů, které ji podporují, v základu zapnutá, ale lze ji vypnout. Pokud je funkce zapnutá, procesor může spustit na každém jádře dvě vlákna. Tato technologie umožňuje nárůst výkonu, pokud jednotlivá vlákna čekají a dají se překrýt. Hodí se taktéž pro spuštění dvou různých aplikací, které využívají jiné funkční jednotky. Hyper-Threading může i prodloužit dobu vykonání programu. Zvláště pokud obě vlákna pracují stejně, potřebují ke svému běhu stejné prostředky a nepotřebují obsluhu mnoha výpadků. Paměť cache je oběma vlákny sdílená, a tak je při Hyper-Threading větší pravděpodobnost výpadku v těchto sdílených cache, než kdyby pracovalo pouze jedno vlákno.

## 5.4 Aplikační rozhraní pro paralelizaci

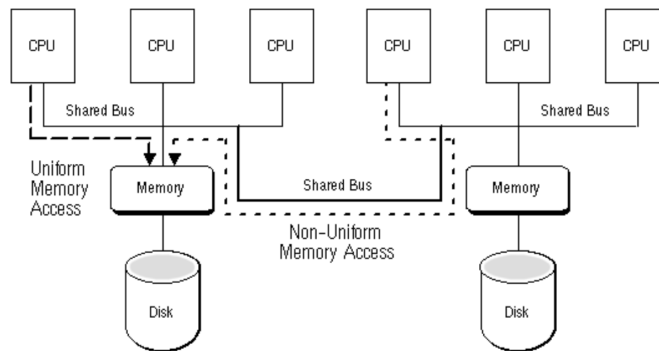
Co do realizace paralelního zpracování dat, existují dva abstraktní modely rozdělení úlohy programátorem mezi více výpočetních jednotek. Jedná se o model sdíleného adresního prostoru SAS (z anglického Shared Address Space) a model zasílání zpráv (Message Passing). Oba tyto modely mohou být praktikovány na běžných paralelních systémech. V případě potřeby mohou být chybějící prostředky daného modelu emulovány.

První, jak plyne už z názvu, spoléhá na jeden adresní prostor, kam mají přístup všechny jednotky. V paměti existuje typicky jedna sada dat a procesní jednotky k ní přistupují. Problémy komunikace a synchronizace mezi jednotkami jsou řešeny zápisem do tohoto paměťového prostoru. Programování tohoto systému je pro programátora snazší, jelikož zápis vlákna X na pozici N vlákno Y zaznamená. Mohou zde ale nastat problémy. Například pokud vlákno X i Y čte, vykoná nějakou operaci a následně se snaží oba zapsat. Tyto problémy lze řešit synchronizací pomocí zámků.

Existují zde dvě reálné hardware implementace, a to UMA (Uniform memory address) a NUMA (Non-UMA). U prvního tvrdíme, že doba přístupu všech procesních elementů k libovolným datům, která nejsou v cache, trvá stejně dlouho. Příkladem můžou být vícejádrové procesory, kde k paměti RAM mají všechny procesy stejný přístup. NUMA architektura pak disponuje rozdílným časem přístupu k různým datům v paměti. Například dvouprocesorový systém, kde každý procesor má vlastní paměť a vzájemně komunikují přes sdílenou linku. Pokud procesor A potřebuje přistoupit do paměti A, je cena/doba přístupu dána pouze dobou komunikace s pamětí. Pokud však procesor A potřebuje data z paměti B, je výsledná cena složena z komunikace s procesorem B a přístupem k paměti B.

NUMA systém bývá většinou složen z výpočetních uzlů. Zachování UMA při rostoucím počtu CPU v systému je problematické kvůli rozptylu pamětí od CPU. NUMA systém je snadnější na rozšiřitelnost. Přístup k lokální paměti může zůstat nízký. Problematické je programování a ladění takového systému. Je potřeba výpočetní úlohu správně rozdělit mezi uzly systému a zajistit efektivní výměnu dat. Stejně tak u rozsáhlejších systémů a úloh je vhodné hlídat cenu na vzájemnou komunikaci. Tato cena se může lišit podle vzdálenosti v dané propojovací topologii.

S programováním na architektuře UMA pomáhá efektivně rozhraní OpenMP (Open Multi-Processing) [7]. Jedná se o aplikační rozhraní usnadňující tvorbu vícevláknových aplikací. Toto rozhraní specifikuje základní prostředky pro práci s vlákny. Například bariéry pro synchronizaci vláken. Vlákna mohou v průběhu vykonávání vznikat i zanikat dle potřeby a nejsou limitovány počtem vláken reálně běžících na procesoru v jednom okamžiku.



Obrázek 5.3: Příklad neuniformního přístupu do paměti.

Kooperaci vláken při plnění nějakého úkolu lze rozdělit na dva typy. Oba mají podporu v OpenMP. Prvním je paralelní vykonávání programu, kdy si vlákna rozdělí data a vykonávají stejnou činnost. Typicky jsou takto rozděleny smyčky *for* a toto rozdělení je označováno jako *work sharing*. OpenMP předpokládá, že doba vykonání každé iterace je vždy stejná. Ne vždy je to pravda. V takových případech je vhodné informovat překladač, jakým způsobem má rozdělovat iterace daným vláknům. Statické plánování přidělí každému vláknům stejně velký blok iterací. Dynamické následně přidělí taktéž každému vláknům stejně velký blok. Ten je ale typicky výrazně menší než u statického plánování a po prvním přidělení zůstávají bloky – iterace doposud nepřidělené. Pokud je práce mezi iteracemi náhodně rozdělená, nemusí na konci zpracování vlákna čekat tak dlouho jako u statického. Přidělování bloku iterací vláknům není zadarmo. Čím častěji přicházejí vlákna pro další bloky, tím více se režie plánování projevuje na době zpracování. Třetím možným způsobem plánování iterací je řízené přidělování (*guided*). Pokud je použito, první vlákno obdrží největší blok iterací a každé další o něco menší. Tento způsob se vyplatí, pokud se práce mezi iteracemi mění lineárně.

Druhým je rozdělení programu na části, sekce, které lze vykonávat nezávisle. Různá vlákna tak procházejí rozdílné části zdrojového kódu. Vhodné rozdělení kódu na sekce vyžaduje, aby mezi těmito sekcemi nebyly datové závislosti. Je logické, že pokud jedno vlákno připravuje data, nemůže zároveň tato data zpracovávat.

Pro používání OpenMP slouží pragmata překladače (`#pragma omp ...`), která zpracovává kompilátor. Od verze 4.0 již OpenMP disponuje i podporou pro vektorizaci. To značně usnadňuje práci programátora.

Jednou s častých uloh při výpočtech je redukce. Tato operace provádí stejnou operaci nad množinou, typicky polem, dat. Výsledek je ukládán do jediné proměnné. Pokud by byla redukce dělana iterativně a v každé iterace by k výsledku byl přidán další člen, byla by časová složitost operace redukce  $O(N)$ . Pro paralelní řešení existuje algoritmus využívající stromové struktury, provádějící redukci v  $O(\log N)$ . Díky OpenMP nemusí programátor tuto operaci programovat sám. Nad vektorizovanými i nad paralizovanými useky kódu je možné vložit klauzuli OpenMP *reduction(op, variable)*. Překladač pak vloží redukci do kódu za programátora sám.

Dalším rozhraním, které lze využít pro paralelizaci, je MPI (Message Passing Interface). To se stará o zaslání zpráv mezi procesory, tedy druhý zmíněný abstraktní model. Hodí se zejména pro architektury s distribuovanou pamětí. Disponuje podporou pro zaslání zpráv složených z běžných i uživatelem specifikovaných datových struktur. Obsahuje podporu pro kooperaci ve skupinách. MPI má dán počet procesů pracujících na úloze již při začátku

programu. Vlákna se mohou ukončit v různou dobu, všechny však o tomto ukončení musí informovat ostatní. Pokud procesy musí zasílat informace ostatním procesům, lze tak učinit právě v daných skupinách přičemž vlákna jsou vždy minimálně v globální skupině všech procesů. Také je přítomna podpora pro vykonávání redukce.

Obě rozhraní mohou být aplikovány na obou architekturách. Použití OpenMP na NUMA může mít negativní vliv na efektivitu. Využití těchto rozhraní umožňuje programátorovi soustředit se přímo na danou aplikaci.

## 5.5 Grafické karty-CUDA

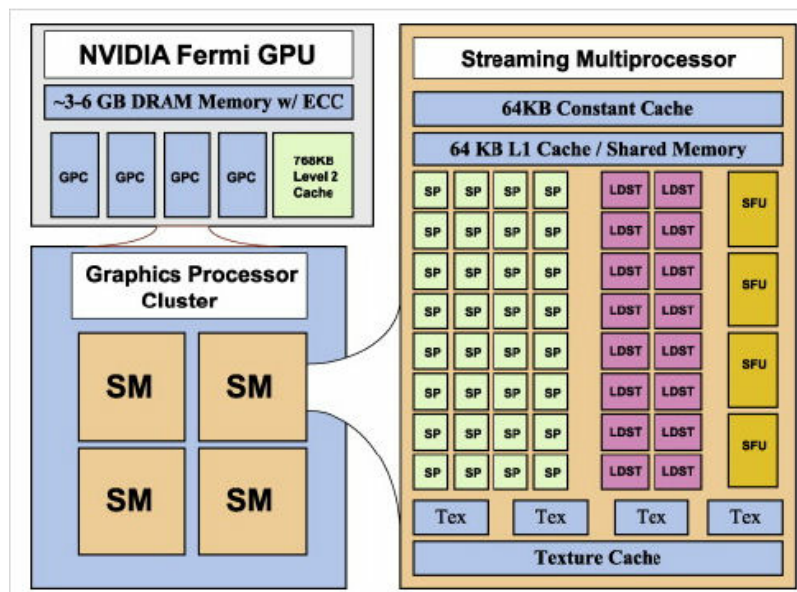
Architektura grafických karet byla dlouhou dobu zaměřena pouze pro využití v oblasti vykreslování. Původní zřetězení jednotlivých hardwareových jednotek bylo fixní. Grafická karta měla speciální obvody pro sestavení primitiv scény, převážně vertexu, jejich rasterizaci a mapování pixelů do výsledného obrazu na výstupní zařízení. S postupem času byla patrná různá potřeba grafických aplikací, kde některé potřebovaly více vertexových a jiné pixelových jednotek. Vzniklo tedy programovatelné propojení tvořené uniformními jednotkami, kterým programátor přiřadil buď funkci vertex, nebo pixel shaderu. Dnešní grafické karty mohou volit z ještě větší sady shaderu, jako například compute či geometry. Velké množství univerzálních výpočetních jednotek umožňuje využití ke zpracování i jiných úloh než je vykreslování obrazu.

Každá generace grafických karet se nějak liší od předchozích. Může se jednat o změnu ve způsobu plánování či rozložení funkčních jednotek. Grafická karta je vždy složena z několika streaming procesorů. Ty obsahují několik CUDA jader, které mohou zpracovávat buď celé číslo, nebo float/double. Dále obsahuje množství Load/Store jednotek starajících se o práci s pamětí. Existují zde i SFU (Super Functional Unit), mající za cíl vykonávat iterační operace (odmocnění, sinus) v pevném počtu kroků, a tím se sníženou přesností. Poměr mezi CUDA jádry, L/S jednotkami a SFU se liší v rámci architektury.

Streaming procesory obsahují taktéž registrová pole, sdílenou (on-chip) paměť a různé druhy cache. Pomocí CUDA lze využívat jednotlivé výpočetní jednotky grafické karty. CUDA pracuje jako SIMT (Single Instruction Multiple Threads), kde je jedna instrukce vykonávána typicky nejméně 32 výpočetními jednotkami současně. Těchto 32 synchronně zpracovávaných vláken je označováno jako WARP. Je možné pracovat s méně vlákny, ale řešení je obdobné jako u SIMD vektorizace, kdy se vykoná všech 32 synchroních operací, ale použije se menší množství výsledků. Pro zanedbání latence paměti je potřeba mít velké množství vláken, tím mohou být paměťové operace překryty výpočtem.

Obrázek 5.4 demonstruje cenu použití jednotlivých částí procesoru. Na každém multiprocesoru je 32 vláken. Najednou však může načítat či ukládat data jen polovina z nich. To je jeden z důvodů, proč je potřeba mít co nejvyšší numerickou intenzitu výpočtu. Numerickou intenzitou je označován počet operací vykonaných nad jedním bajtem dat.

Superfunkcionální jednotky (SFU) jsou na multiprocesoru přítomny proto, aby výpočet iteračních úloh byl proveden každým vláknem za stejnou dobu. Tyto úlohy běžně pracují v cyklu a při každé iteraci zpřesňují svůj výsledek. Když je nalezena požadovaná přesnost výpočet je ukončen. Pro různé vstupy je počet potřebných iterací předem neznámý. Pokud některá vlákna ve WARP naleznou výsledek a ostatní ne, musí řízení multiprocesoru tato vlákna rozdělit. Superfunkcionální jednotky mají pevně definovaný počet opakování pro



Obrázek 5.4: Zjednodušené schéma grafické karty Nvidia architektury "Fermi". Jeden multiprocessor je vybaven 32 funkčními jednotkami pro celočíselné a desetinné výpočty (v jednoduché přesnosti), šestnácti load/store jednotkami pro práci s pamětí a čtyřmi superfunkčními jednotkami pro iterační výpočty jako jsou dělení či funkce sinus.

jednotlivé funkce, cenou za to je snižená přesnost pro některé hodnoty. Přesnější informace o této přesnosti je možné dohledat ve specifikaci. <sup>1</sup>

## 5.6 Organizace dat v paměti

Značný vliv na výkon programu může mít i organizace dat v paměti. Jak bylo zmíněno výše, výpadky v paměti cache jsou považovány za drahé, a i když se jim nelze zcela vyhnout, je vhodné je minimalizovat. Správná organizace dat může urychlit přípravu těchto dat do funkčních jednotek, zvláště do vektorových registrů nebo vstupu do procesních jednotek grafické karty. U SIMD zpracování se dá hovořit o záměně z pole struktur na strukturu polí. Logická organizace dat do pole struktur, například pole sta bodů v prostoru kdy každý má souřadnice  $x$ ,  $y$  a  $z$ , je přehledná pro programátora. Tato struktura je dostatečně krátká, aby se mohla vejít do jediného bloku cache. Předpokládejme, že do tohoto bloku se vlezou tři hodnoty. Pokud potřebujeme změnit pozici jednoho bodu o  $\delta y$ , přistoupíme ke struktuře a modifikujeme  $y$ . Dochází zde maximálně k jednomu výpadku v paměti cache.

Nyní si představme, že potřebujeme změnit sto bodů a u každého změnit hodnotu  $y$  o  $\delta y$ . Je potřeba přistoupit k  $(100 * 3) / 4 = 75$  bloků cache. Tato organizace dat lze zaměnit za případ, kdy máme v paměti jednu strukturu, která je složena ze tří polí  $x[]$ ,  $y[]$  a  $z[]$ , každé délky 100. V případě, že je potřeba modifikovat hodnoty  $y$  je potřeba načíst pouze tyto hodnoty a počet potřebných bloků cache je  $100 / 4 = 25$ . Tedy jedna třetina oproti poli struktur [5].

Dalším vhodným postupem je zarovnání dat v paměti. Při zarovnání na hranici  $N$ , kompilátor vkládá položky vždy na adresu, která je násobkem  $N$ . To následně umožňuje efektivnější přístup.

<sup>1</sup><http://docs.nvidia.com/cuda/cuda-c-programming-guide/#intrinsic-functions>

Třetí a poslední technika používaná pro zefektivnění práce s pamětí je *padding* neboli vycpání. Velmi užitečné je právě u vektorizace. Pokud existuje pole o  $M$  prvcích a do vektorizačního registru se těchto prvků vejde  $N$ , je operace provedena tolikrát kolikrát je registr  $N$  zcela naplněn. Pro zbyvajících  $M$  modulo  $N$  prvků je provedena skalární operace. Právě v těchto případech se využívá padding. Data o délce  $M$  jsou doplněna neutrálními prvky, nulami či jinými tak, aby výsledná délka dat byla násobkem  $N$ . Místo vykonání několika skalárních operací na závěr vykonávání vektorizace je tak provedena jedna vektorová operace, z níž je využito jen několik výsledků. S daty v oblasti padding se samozřejmě nepracuje a neměla by ovlivňovat ostatní výpočty.

## Kapitola 6

# Optimalizace slabých míst

Výše byl popsán princip simulace i její implementace. Jednotlivé kroky simulace je potřeba vykonávat postupně ve správném pořadí. Cílem je, aby práce uvnitř každé iterace proběhla co nejrychleji. K tomu byla využita technika vektorizace na procesoru spolu s paralelizací. V této kapitole budou zobrazeny části kódu, který je upravován. Jedná se hlavně o části funkce pro aktualizaci reakcí působící na jednotlivé body *brick* elementů. Celou funkci v jednotlivých fázích vývoje je možné nalézt v exportovaném repozitáři, který je obsahem příložného DVD.

### 6.1 I/O

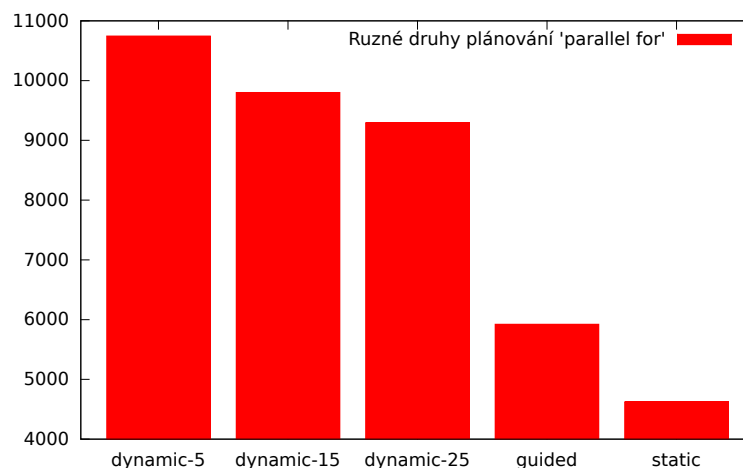
Při některých smyčkách simulace je potřeba uložit stav simulace pro výsledný P-CMOD graf. Toto vypisování se děje jednou za  $N$  iterací podle dané konfigurace. Při standartním testování simulace bylo použito 10 000 000 iterací a po každých 10 000 byl proveden zápis do souboru dvojicí příkazů *fprintf()* a *fflush()*. Celkově se jedná o tisíce zápisových operací.

Při odstranění požadavku na okamžitý zápis na disk z hlavní smyčky dochází pouze k zápisu do paměti. Samotný zápis na disk je proveden až po dokončení simulace. Touto modifikací byla simulace zrychlena o 5 procent. Přesněji z 29 030 sekund na 27 815 sekund.

### 6.2 Paralelizace konečných elementů

Práce vykonávána nad jednotlivými konečnými elementy – *brick* je stejná. V modelu je přibližně 2 000 *brick*. Testovaný referenční model jich měl 2 310. Výpočet reakcí působících na *brick* zahrnuje načtení matice tuhosti, která je vlastní každému *brick*. Načtení výchozích pozic bodů, což je neměnná hodnota přečtená při inicializaci ze vstupního souboru. Dále je pro výpočet potřeba aktuální pozice bodů, která se během simulace mění. V rámci jedné iterace je však stálá až do určení nových pozic. Vstup funkce pro výpočet reakcí je během výpočtu jedné iterace nezávislý na zpracování ostatních elementů. Tím je vhodný pro distribuovaný výpočet.

Pro paralelní zpracování byla využita specifikace OpenMP. Nad smyčku *for* zpracovávající výpočet reakcí uvnitř *brick* bylo přidáno *#pragma parallel for*. To říká kompilátoru, že si programátor přeje rozdělit výpočet mezi vlákna procesoru. Každou iteraci smyčky je tak možno vykonat jiným vláknem. O počtu vláken rozhoduje nastavení OpenMP. Ve výchozí konfiguraci je každé vlákno mapováno na jedno fyzické jádro procesoru. Tedy existuje tolik vláken kolik je jader. Pokud si programátor přeje, je možné spustit i více vláken, než je



Obrázek 6.1: Délka běhu programu na osmi jádrech s různým nastavením plánování iterací. NevektORIZOVÁNO.

k dispozici jader, a ty se na jádře střídají. Při našem testování byla použita dvojice procesorů Intel Xeon E5-2665, z nichž každý má k dispozici 8 jader.

Kromě označení smyčky zmíněným `#pragma` je také důležité zvolit správnou strategii plánování. V modelu přítomných 2 310 elementů je potřeba přiřadit několika málo vláknům. Existuje několik možností, jak tato vlákna přidělit. Graf 6.1 zobrazuje výsledek běhu na osmi vláknech s různým nastavením plánování.

Dynamické plánování znamená, že každé vlákno dostane na starost určitý počet iterací, v OpenMP označovaný jako *chunk*. Po jejich vykonání dostane opět maximálně takto velký *chunk* až do doby, kdy jsou zpracovány všechny iterace. Na obrázku 6.1 jsou vyzkoušeny možnosti s velikostí 5, 15 a 25 iterací. Tyto časy s rostoucí velikostí části *chunks* klesají. Může za to klesající režie na plánování, zvláště klesající počet žádostí o nové *chunk*. Toto dynamické plánování by jistě bylo pro další rostoucí velikosti *chunk* ještě rychlejší než prezentované hodnoty. Pro ilustraci trendu jsou tyto tři hodnoty dostačující.

Další možnost *guided* je ještě nižší, opět z důvodu klesající režie. V tomto případě je každý *chunk* přidělený vláknům dynamicky menší než předchozí. Tato volba se hodí více k úlohám, kdy čas na jednu iteraci lineárně klesá.

Nejlépe vyšla možnost statického plánování, kde jsou iterace rozděleny jen jednou a každému vláknům je přiřazen souvislý blok *počet iterací/počet vláken*. To má také pozitivní vliv na cache, jelikož *brick* elementy definované za sebou často sdílí některé body. Tato skutečnost byla spozorována u většiny modelů, ale nemusí být pravidlem.

Na vzorku kódu 6.1 je vidět část původního kódu doplněná o příkaz `#omp parallel for`. Tato úprava přinesla v původní verzi na jednom procesoru Intel Xeon E5-2665 zrychlení z 27 815 sekund na 4 611 sekund. Tedy na osmi jádrech je toto řešení 6x rychlejší. Podrobnější přínos této úpravy je rozebrán v sekci 7.3

### 6.3 Zaměna Double za Float

Pro dosažení většího zrychlení je možné v některých úlohách možné využít místo proměnných typu *double* typ *float*. Někteří programátoři neznají standardu IEEE 754 [20] se mohou domnívat, že do typu *double* lze uložit menší čísla. To je sice pravdou, ovšem rozdíl ve

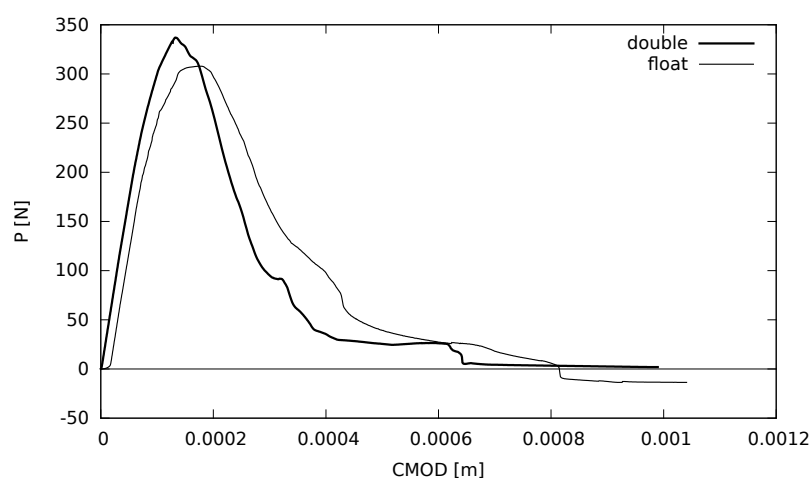


```

#pragma omp parallel for schedule(static)
for (int i = 0; i < numberOfBricks; i++) {
    ...
    double reaction = 0;
    for (int k = 0; k < 24; k++)
    {
        reaction += matrixOfStiffness[i * 24 * 24 + j * 24 + k]
                    * deformations[k];
    }
    ...
}

```

Listing 6.1: Paralelní verze původního kódu



Obrázek 6.2: Rozdíl výsledků float a double verze programu

velikosti exponentu jsou pouze tři bity. *Float* obsahuje 8 bitů pro exponent, kdežto *double* jich obsahuje 11. Hlavní rozdíl mezi těmito typy je v počtu významových bitů, kde *float* má pouze 24 bitů, zatímco *double* jich má 53.

Tato záměna by znamenala, že do jednoho bloku cache by bylo možno umístit dvojnásobný počet hodnot. Navíc, pokud by se hodnoty které simulace zpracovává daly vyjádřit na 24 významových bitech, bylo by možné technikou vektorizace počítat současně dvojnásobný počet hodnot. Lze předpokládat odchylku v těchto výsledcích. Pokud by tato odchylka byla dostatečně nízká, je možno ji přijmout jako cenu za dostatečné zrychlení.

Bylo proto nejdříve ověřeno, jaký vliv má záměna těchto datových typů. Jak ukazuje obrázek výsledky neodpovídají.

Rozdíl od správného grafu *double* a chybného *float* je značný. Na třech usecích lze vidět problémy s modifikovanou verzí. Hned na počátku křivky si lze povšimnout rozdílu v tom, že roste vzdálenost CMOD a není akumulováno žádné zatížení na podepřeném bodě. Dalším je maximum grafu, které je výrazně nižší, než by mělo být. Konec grafu jde pak do záporných hodnot, což krom toho, že neodpovídá referenci, ani zdaleka neodpovídá principu metody.

Tyto rozdíly jsou způsobeny právě nízkou přesností datového typu *float*. Výsledné reakce jsou v počátku natolik malé, že jejich vliv se na 23 významových bitech takřka neprojeví. Při

```

for (int j = 0; j < 24; j++) {
    double reaction = 0;
    for (int k = 0; k < 24; k++)
    {
        reaction += matrixOfStiffness[i * 24 * 24 + j * 24 + k]
            * deformations[k];
    }
}

```

Listing 6.2: Původní úzké hrdlo

dostatečné akumulaci a rozběhnutí simulace jsou už reakce dostatečné, chybějící hodnoty z počátku simulace již však odsuzují celou simulaci k neúspěchu. Z tohoto důvodu nebylo možné s *float* verzí programu pracovat.

## 6.4 Vektorizace

Dalším krokem pro efektivnější využití procesoru bylo zapojení vektorizačních jednotek procesoru. Určení reakcí v modelu je výpočetně nejnáročnější process. Každou iterací jsou zpracovány všechny konečné elementy *brick*. Každá iterace vykonává stejné operace nad jinými daty. Proto je vhodná pro přístup single instruction multiple data (SIMD). Potenciální zrychlení při nasazení vektorizace závisí na délce vektorizačních registrů. Zde uvažovaná simulace potřebuje pracovat s čísly v plovoucí řádové čárce dvojté přesnosti, double precision (DP). To vylučuje použití rozšíření SSE, které disponuje pouze možností pracovat se single precision (SP) čísly. SSE2 registry jsou délky 128bitů, a tak dokáží současně zpracovat dvě hodnoty. Následující rozšíření AVX již obsahuje 256 bitové registry a dokáže zpracovávat 4 double precision současně.

Že se má vektorizovat, lze informovat překladač zdrojového kódu přepínačem. Překladač poté ty části kódu, které uzná za vhodné vektorizuje. Ne vždy jsou části kódu, které chce programátor vektorizovat, správně překladačem vyhodnoceny. Pro účely této práce bylo zvoleno specifikace OpenMP. Ta specifikuje pragmata pro překladač, kterými může programátor o svých úmyslech lépe informovat.

Nejvnitřnější smyčka procesu pro výpočet reakcí může být viděna na části kódu 6.2. Tato smyčka byla označena pragmatem pro využití SIMD. Zároveň bylo informováno o tom, že všechny mezivýpočty v rámci iterace jsou shromažďovány do jediné proměnné. Upravený kód je vidět na části 6.3. Tato úprava při použití AVX zrychlila běh referenčního modelu z 27 815 sekund na 12 827 sekund. Tedy vektorizovaná verze je 2,16 krát rychlejší než předchozí. Teoretické zrychlení při zpracování 4 prvků současně by mělo být blíže k hodnotě 4.

Vektorizace zde nedosahuje plného potenciálu. Může za to z části princip zpracování redukce do jedné proměnné, který nelze vykonat v jednom taktu. Typicky se při redukci v každém cyklu sečtou páry hodnot, dokud nezůstává jen jedna výsledná. Druhý důvod pro nedosažení plného potenciálu zrychlení AVX pro hodnoty typu double je malé množství práce ve vektorizované smyčce. Data po přípravě do vektorizačních registru jsou okamžitě zpracovány a ukládány. Není tak možné překrývat latenci paměti.

```

for (int j = 0; j < 24; j++) {
    double reaction = 0;
    int base = i * 24 * 24 + j * 24;

    #pragma omp simd reduction(+: reaction)
    for (int k = 0; k < 24; k++) {
        val = stiffness[base + k] * deformations[k];
        reaction += val;
    }
}

```

Listing 6.3: Vektorizovaná verze úzkého hrdla

## 6.5 Rozbalení smyčky

Vektorizovaná smyčka z předchozí sekce obsahuje poměrně málo práce. Elementy je potřeba nahrát do vektorizačních registrů a po zpracování opět odeslat do příslušného místa v paměti. V tomto případě se výsledek ukládá na jediné místo, proto část pro uložení není tak výrazná jako samotné načtení. Pro zvýšení práce uvnitř smyčky a vylepšení procesu zpracování byla vnější smyčka obalující vektorizovanou část rozbalena [3]. Bylo zvoleno rozbalení  $3\times$ , což částečně zpřehledňuje kód, který doposud pracoval se všemi koordináty 3D prostoru stejně.

Tato úprava poskytla výrazné zrychlení běhu. V sekvenční verzi, tedy na jednom jádře, se čas pro simulaci dostal na 7 584 sekund. To je 1,69 krát rychlejší než předchozí vektorizovaná verze. Celkové zrychlení oproti původnímu času je 3,67 krát. Potenciální zrychlení vektorizovaného kódu při použití AVX by se mělo blížit k hodnotě 4. Tím, že ne veškerá práce simulace je vektorizována nelze, této hodnoty plně dosáhnout.

## 6.6 Odstranění režie tvorby vláken

Konečná úprava se týká verze s použitím paralelizmu. Při analýze pomocí programu VTune bylo zjištěno, že paralelní verze tráví nejvíce času vykonáváním funkce *clone()*, která vytváří nová vlákna a je do programu přidána na základě *#pragma omp parallel for*. Bylo tedy rozhodnuto, že se vlákna vytvoří pouze jednou, budou pracovat synchronizovaně, tam kde stačí jedno vlákno, tedy budou vykonávat stejnou práci. Pouze v částech, jež lze zpracovat paralelně, typicky vnější smyčky *for* pro *brick* nebo *bodies*, budou označeny *#pragma omp*

Tabulka 6.1: Tabulka doby simulace v různých stadiích optimalizace. Každý řádek kromě své optimalizace obsahuje i úpravy z řádků předchozích.

Počet jader	1	2	4	8	16
Original	27 815s	14 545s	7 738s	4 611s	4 203s
Vektorizováno (AVX)	12 827s	7 014s	4 023s	2 744s	3 291s
Rozbalení smyčky	7 584s	4 371s	2 673s	2 065s	2 681s
Odstranění režie tvorby vláken	7 531s	3 931s	2 124s	1 320s	1 903s

```

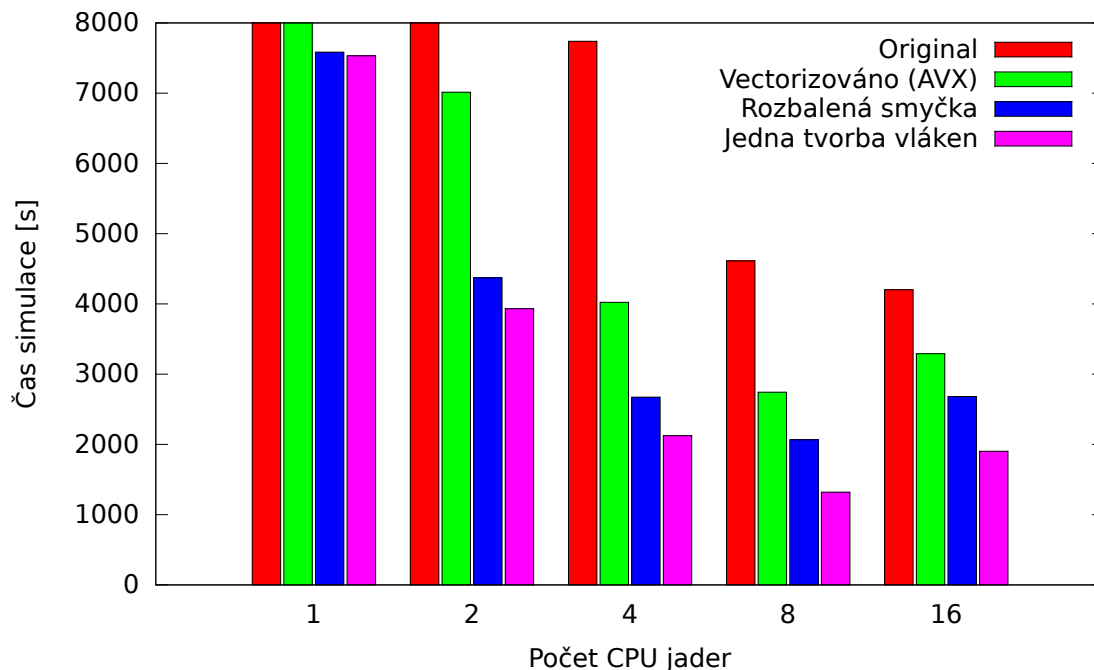
for (int j = 0; j < 24; j+=3) {
    double reactionX = 0;
    double reactionY = 0;
    double reactionZ = 0;

    int baseX = i * 24 * 24 + j * 24;
    int baseY = i * 24 * 24 + (j+1) * 24;
    int baseZ = i * 24 * 24 + (j+2) * 24;

    #pragma omp simd reduction(+: reactionX, reactionY, reactionZ)
    for (int k = 0; k < 24; k++) {
        valX = stiffness[baseX + k] * deformations[k];
        valY = stiffness[baseY + k] * deformations[k];
        valZ = stiffness[baseZ + k] * deformations[k];
        reactionX += valX;
        reactionY += valY;
        reactionZ += valZ;
    }
}

```

Listing 6.4: Rozbalení smyčky pro odstranění režie smyčky a překrytí načítání hodnot z paměti výpočtem.



Obrázek 6.3: Časy simulací na různém počtu jader a s různou úrovní optimalizace. Rozsah osy Y je oříznut na 8 000 sekundách. Přesné hodnoty jednotlivých běhů lze vyčíst z Tabulky 6.1

*for*. Verze s rozbalenou smyčkou na osmi jádrech dosahuje rychlosti 2 065 sekund. Při odstranění režie je tento čas na stejném počtu jader 1 320 sekund. Tedy 1,5 krát rychlejší.

Toto zrychlení není pouze zásluhou odstranění režie. V předchozích upravách byly paralelně zpracovány pouze *brick* elementy. Při zpracovávání celé simulace více vláken bylo přidáno paralelní zpracování i k aktualizaci bodů. Tato aktualizace není tak náročná jako zpracování *brick*, její paralelizace však taktéž přispěla k tomuto zrychlení.

## Kapitola 7

# Analyza výsledné verze

Tato kapitola prezentuje rozdíly oproti původní verzi popsané v Kapitole 4 a také hlouběji ukazuje a diskutuje výsledky Kapitoly 6.

### 7.1 Optimalizované využití procesoru

Stejně jako v Kapitole 4.2 jsou zde prezentovány metriky naměřené pomocí PAPI rozhraní. Tentokrát jsou to výsledky výsledné optimalizované verze v Tabulce 7.1. Tabulka 4.1 je zde uvedena znovu jako Tabulka 7.2 pro pohodlí čtenáře této práce při srovnání dosažených výsledků.

Tabulka 7.1: Tabulka výkonových parametrů optimalizované implementace

	Jedno jádro	Medián z osmi jader	Osm jader
MFLOPS	3 458,38	2 806,58	22 433,6
Vypadky v L2 cache	30,1 %	33,75 %	33,63 %
Výpadky v L3 cache	0,0997 %	3,59 %	3,64 %
Cykly bez instrukce	1,89 %	0,882 %	1,019 %

Tabulka 7.2: Tabulka výkonových parametrů původní implementace.

	Jedno jádro	Medián z osmi jader	Osm jader
MFLOPS	936,36	794,82	6 353,22
Vypadky v L2 cache	6,880 %	8,600 %	8,900 %
Výpadky v L3 cache	0,0992 %	8,845 %	8,837 %
Cykly bez instrukce	0,0225 %	0,234 %	0,289 %

V tabulce si lze na první pohled povšimnout navýšení výkonu v plovoucí řádové čárce. Jmenovitě původní hodnota při sekvenční verzi 936,36 MFlops byla navýšena na 3 458,39 MFlops. To je 3,58 krát více. U mediánu a celkového výkonu na osmi jádrech je toto navýšení obdobné. Teoretický výkon procesoru na jádro je 19,2 GFlops. Původní verze dokázala tento potenciál využít z 4,68 procent. Optimalizovaná verze využívá 18,01 procent potenciálu procesoru. Stále lze v hodnotách sledovat pokles výkonu na jádro při přechodu ze sekvenční

na paralelní verzi. Tato skutečnost je opět způsobena sdílením L3 cache mezi jádry. Vzniká potřeba udržovat koherenci paměti a tím roste počet výpadků.

Zajímavé je taktéž navýšení počtu výpadků v L2 cache. To je možné sledovat již v sekvenční verzi, kde z původních 6,88 procent došlo k navýšení na 30 procent. To je způsobeno tím, že jádro procesoru zpracovává rychleji než je procesor schopen data připravovat. Počet výpadků v L2 cache vzrostl jen nepatrně v sekvenční verzi. Jak bylo zmíněno výše, do této cache je možné umístit značnou část úlohy a při využití pohým jedním jádrem je procesor schopen udržovat zde potřebná data.

V počtu výpadků u paralelní verze došlo taktéž k zvýšení v L2 cache. Rozdíl oproti sekvenční verzi je kolem tří procent a lze bezpečně usoudit, že za něj mohou techniky pro udržování koherence dat stejně jako v původní implementaci. U počtu výpadků v L3 cache došlo k snížení oproti původní implementaci. Díky provedení rozbalení smyček má nyní procesor možnost efektivněji predikovat, jaká data budou potřeba v další iteraci a může je efektivně přednačítat.

U poslední měřené metriky, počtu cyklů bez připravení instrukce k vykonání, lze pozorovat nárůst. Toto čekání je způsobeno čekáním procesoru na paměť a přípravu dat. Jak v sekvenční, tak paralelní verzi došlo k růstu těchto hodnot. Celkový čas simulace je však výrazně nižší, a proto je počet takovýchto čekání v celkovém počtu znatelnější. Počet vykonaných instrukcí pro 100 000 iterací simulace původního kódu byl  $1\,002 \times 10^9$ . U optimalizované verze byl počet vykonaných instrukcí na stejně velkou úlohu  $384 \times 10^9$ , tedy 38% původního počtu.

Tabulka 7.3 porovnává rychlost za jednotku u původní a optimalizované. Tyto výsledky přímo vyplývají z Tabulky 6.1. Čas na iteraci je lineárně závislý na velikosti zpracovávaného modelu. Z tohoto důvodu je v tabulce čas na element, z něž lze v budoucnu odvodit dobu iterace různých velikých modelů.

Tabulka 7.3: Tabulka rychlosti zpracování iterací původní a optimalizovanou implementací. Hodnoty ve sloupci *jedno jádro* připadá na výkon v sekvenční verzi, hodnoty ve sloupci *osm jader* připadají na celkový výkon. Sloupec *Na jádro* ukazuje výkon jednoho jádra, v situaci kdy jsou všechny jádra aktivní.

Verze	Původní			Optimalizovaná		
	Jedno jádro	Osm jader	Na jádro	Jedno jádro	Osm jader	Na jádro
Čas na iteraci	2,782ms	0,461ms	3,688ms	0,753ms	0,132ms	1,056ms
Iterací za sekundu	359	2 168	271	1 327	7 575	946
Čas na element	1,204 $\mu$ s	0,199 $\mu$ s	1,596 $\mu$ s	0,326 $\mu$ s	0,0571 $\mu$ s	0,456 $\mu$ s
Elementu za sekundu	830 564	5 009 759	626 219	3 067 321	17 500 000	2 189 141

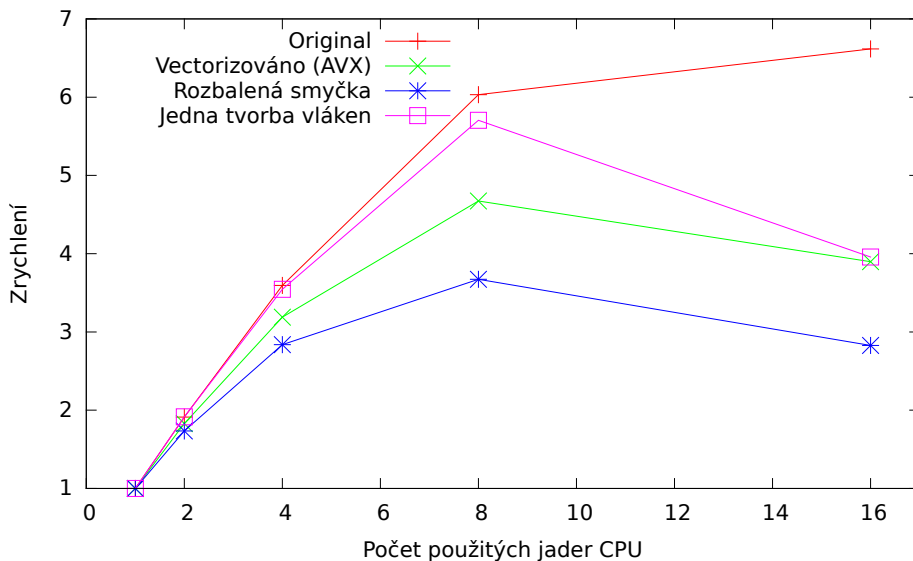
## 7.2 Kritická sekce

Jedním z rysů původní implementace bylo to, že 60 % procesorového času bylo stráveno na jednom jediném řádku kódu. Optimalizovaná verze zde také tráví nejvíce času z celé simulace. Podíl času byl však optimalizací snížen na pouhých 30 % celkové doby běhu simulace. Poměr času, který zde procesor tráví tak výrazně klesl. Můžeme vzít celkový čas na iteraci a určit z něj přímo hodnotu času připadající tomuto úseku kódu. Původní hodnota je 2,782ms, na slabé místo tak připadá přibližně 1,669ms. Sama optimalizovaná sekvenční verze zvládá iteraci za 0,753ms. Z toho v kritické sekci stráví přibližně 0,226ms.

## 7.3 Škalovatelnost

Dalším údajem hodnotícím programové řešení bývá škálovatelnost, viz Kapitola 5.2. V této práci je prezentována pouze silná škálovatelnost daného řešení. Pro realizaci slabé nebyl k dispozici dostatečně různorodý dataset pro naměření slabé škálovatelnosti.

Ideálně by tedy úloha zpracovávaná na  $N$  jádrech měla být  $N$ -krát rychlejší, než stejně velká uloha zpracovávaná jediným jádrem. Reálně se tohoto výsledku dá dosáhnout jen velmi obtížně. Při rozdělení na  $N$  jader přibývá režie na komunikaci. Obrázek 7.1 zobrazuje, jak se schopnost škálování mění v různých stupních optimalizace.



Obrázek 7.1: Škálování simulace v různých stupních optimalizace na superpočítači Anselm.

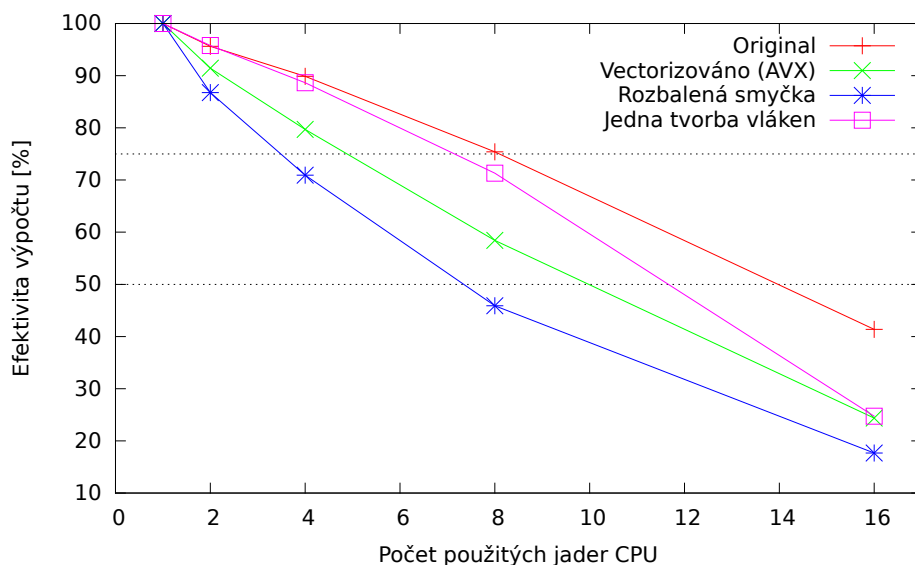
Lze si všimnout, že jediná verze, která při přechodu z 8mi jader na 16 zrychluje, je původní implementace. Ta také v grafu škálovatelnosti vychází nejlépe. Samotný dlouhý čas sekvenční verze a i delší časy paralelních verzí umožňují částečně překrýt komunikaci mezi sokety procesoru. Zrychlení z 6ti na 6,5 však není při přechodu na dva osmijádrové nikterak výhodné. Z ostatních verzí nejlépe škáluje výsledná verze simulace na jednom procesoru. Při přechodu na 16 jader nejznatelněji pociťuje komunikaci mezi sockety.

Dalším parametrem paralelního výpočtu je efektivita paralelizace. Ta opět vychází z ideálního předpokladu, že zpracování  $N$  jádry by mělo být  $N$ -krát rychlejší než zpracování jedním. Může se ukázat efektivnější spustit dvě simulace s různými parametry, každé na jednom jádře nebo procesoru, než jednu simulaci na dvou jádrech či procesorech. Zde je potřeba mít na paměti sdílenou cache na procesoru, která tuto metriku zkresluje. Směrodatná je hlavně při komunikaci několika procesoru mezi sebou.

Při využití celého procesoru, tedy osmi vláken, se efektivita paralelizace původní verze pohybuje kolem 75 procent. Výsledná optimalizovaná implementace těsně nad 71 procent. Při dvou procesorech je efektivita původního řešení 41 procent a optimalizovaného 24 procent.

Z výsledků prezentovaných v Tabulce 7.3 jsou patrné nízké časy na iteraci. Procesy musí komunikovat, například z důvodu udržení platných dat v pamětech L1-L3. Latence přístupu do cache [16] na stejném čipu se pohybuje v rozmezí od 15 po 40 ns. Při přístupu na druhý socket se tyto časy pohybují mezi 87 a 130 ns. Časy v citovaném textu byly





Obrázek 7.2: Efektivita paralelního výpočtu na superpočítači Anselm.

měřeny na dvousocketovém systému a dvojici procesorů z rodiny Intel Sandy Bridge. Novější architektury mohou mít časy nižší. Dá se však předpokádat stejný poměr mezi přístupem na čipu a na druhém socketu. Ty jsou ve zde zmíněném článku 3,25 krát delší než přístup na stejném procesoru.

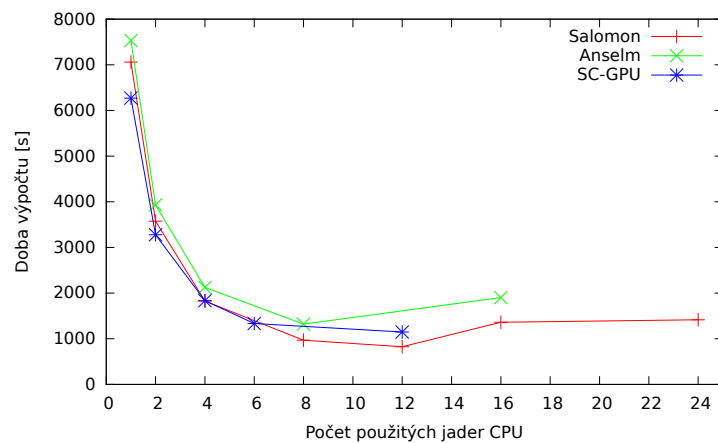
Obrázek 7.1 ukazuje, že i komunikace mezi dvěma procesory osazenými na jedné základní desce představuje pro danou simulaci problém.

Bylo provedeno i testování škalovatelnosti na jiných procesorech. Na Anselmu jsou dvojice osmijádrových procesorů Intel Xeon E5-2665 taktovaných na frekvenci 2,4 GHz. Na těchto procesorech byla aplikace testována přednostně. Dalším procesorem je na superpočítači Salomon dostupný, dvanáctijádrový, Intel Xeon E5-2680v3, taktovaný na frekvenci 2,5 GHz. Třetím otestovaným je pouze šetijádrový Intel Xeon E5-2680v3 s frekvencí 2,4 GHz dostupný na školním serveru SC-GPU1. Všechny tři procesory jsou osazeny na základních deskách s dvěma sokety, takže je možné testovat i přechod mezi UMA a NUMA.

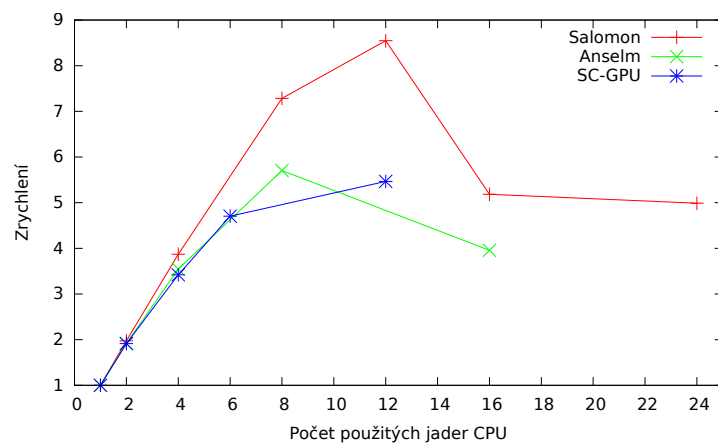
Graf na Obrázku 7.3 zobrazuje dobu potřebnou k simulaci stejné úlohy při použití různého počtu jader. Ty se dost podobají. Je možno si všimnout zlomu v maximálním počtu jader na daném socketu. Procesory na Anselmu mají nejlepší časy pro osm jader. Stejně tak Salomon dosahuje nejlepších výsledků pro 12 jader. U SC-GPU došlo ještě k nevýraznému zlepšení při přechodu na dva sockety. Toto zlepšení lze přikládat tomu, že běžících vláken je relativně málo a zpracovávají větší bloky iterací. Mechanismy pro koherenci cache tak mají více času. Obrázek 7.4 zobrazuje škálování na daném systému. Lze zde vyčíst, že procesory dobře škálují právě na jednom socketu. Posledním grafem této kapitoly je efektivita výpočtu na těchto třech testovaných systémech. Nejeefektivnější škálující je Salomon. Všechny tři systémy jsou při využití jednoho socketu efektivní z více než 70%.

## 7.4 Závislost rychlosti na frekvenci procesoru

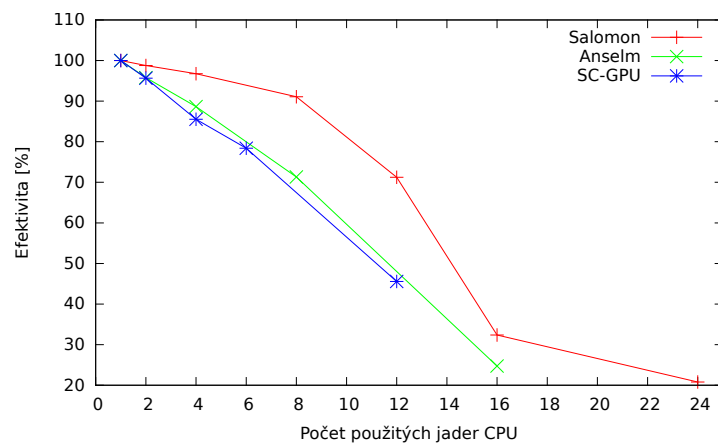
Jedním z limitujících faktorů výpočtu je propustnost hlavní paměti. Ta se během let nezvětšuje tak rychle jako výpočetní výkon procesorů. Pokud propustnost paměti výpočet



Obrázek 7.3: Porovnání času simulace na různých procesorech.



Obrázek 7.4: Porovnání škálování paralelizace na různých systémech.



Obrázek 7.5: Porovnání efektivity paralelizace na různých systémech.

nelimituje, roste s frekvencí procesoru lineárně práce vykonána za čas. Pokud paměť začne omezovat výpočet, je růst zastaven na určité hodnotě vykonané práce. Frekvenci na níž do-

Tabulka 7.4: Tabulka výsledků na jednotlivých systémech při plném využití jednoho procesoru.

Systém	Sekvenční čas	Paralelní čas	Zrychlení	Efektivita
SC-GPU (6 jader)	6 271s	1 333s   22,82min	4,704	78,407%
Anselm (8 jader)	7 531s	1 320s   22,00min	5,705	71,316%
Salomon (12 jader)	7 060s	826s   13,76min	8,547	71,226%

cházi k tomuto zlomu a práce už dále nestoupá je vhodné u daného programu znát. Zvláště u programů s dlouhou dobou běhu je vhodné snížit rychlost procesoru, podtaktovat jej, na tuto frekvenci. Díky tomu lze snížit spotřebu energie i odpadní teplo. Na běh programu to však nemá vliv. Stejně tak, pokud je pro daný program známa maximální efektivní frekvence, je možné pro účel běhu pouze tohoto programu pořídit levnější procesor s touto nižší frekvencí.

Za účelem zjištění této meze výpočtu u optimalizované verze simulace bylo provedeno měření doby běhu pro různé frekvence procesoru. K tomuto měření je potřeba mít administrátorský přístup k systému, na němž je měření prováděno. Nebylo možné vykonat toto měření na referenčním stroji Anselm z IT4Inovations a procesoru Intel Xeon E5-2665. Místo toho bylo provedeno na školním serveru, který má k dispozici obdobný procesor Intel Xeon CPU E5-2620. Tento procesor má pouze 6 fyzických jader. Měření bylo provedeno na frekvencích od 1,2 GHz až po 2,4 GHz. Na Obrázku 7.6 jsou zobrazeny časy tohoto měření a na Obrázku 7.7 je pak výkon v iteracích za sekundu.

Závěrem je, že v rozmezí 1,2 až 2,4 GHz se maximální efektivní frekvence na procesoru Intel Xeon CPU E5-2620 nenachází.

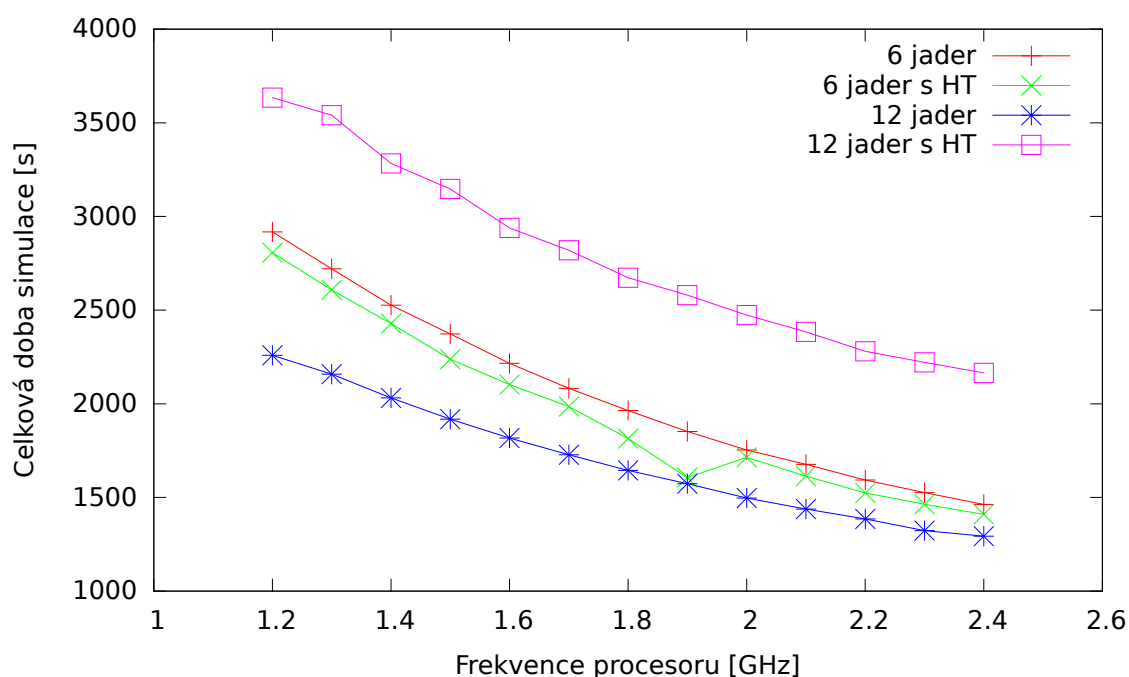
## 7.5 Použití technologie Hyper-Threading

Při testování limitu paměti bylo provedeno i testování se zapnutou funkcí procesoru hyperthreading. Tato technologie je popsána v Kapitole 5.3. Při krokování frekvence na dvojici šesti-jádrových procesorů Intel Xeon CPU E5-2620 bylo ověřeno, že při přechodu na NUMA architekturu dochází k nárůstu výkonu jen minimálně. Při zapojení dvou šesti-jádrových procesorů bez hyperthreadungu je výkon jen o 13 procent vyšší než na jednom procesoru. Tento pokles v efektivitě je způsoben režii komunikace mezi sockety procesoru.

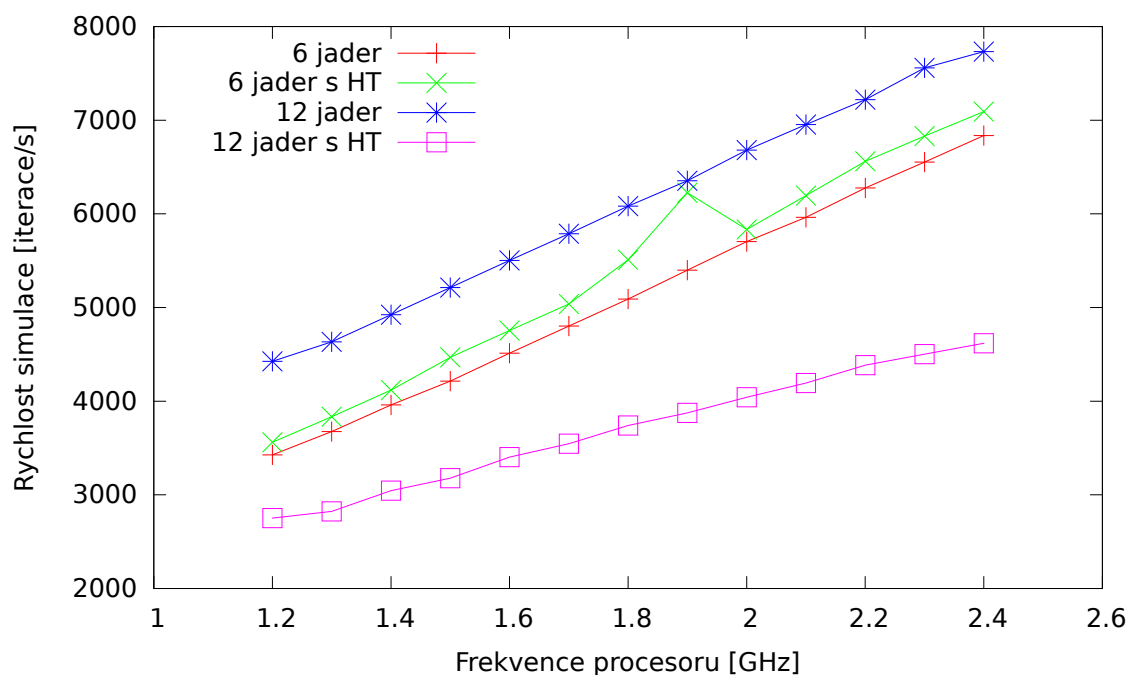
Při zapnutí hyperthreadingu a výpočtu na jednom socketu dochází k mírnému 3 procentnímu nárůstu výkonu. Pokud však je hyperthreading zapnut na dvou socketech dochází poklesu výkonu a to o 33 % oproti verzi na jednom socketu bez hyperthreadingu. Časy na jednotlivých konfiguracích a frekvenci 2,4 GHz shrnuje Tabulka 7.5:

Tabulka 7.5: Tabulka výkonu v počtu iterací za sekundu s a bez hyperthreading

	Bez hyperthreading	S hyperthreading
Jeden procesor(6 jader)	6 835	7 092
Dva procesory(12 jader)	7 733	4 618

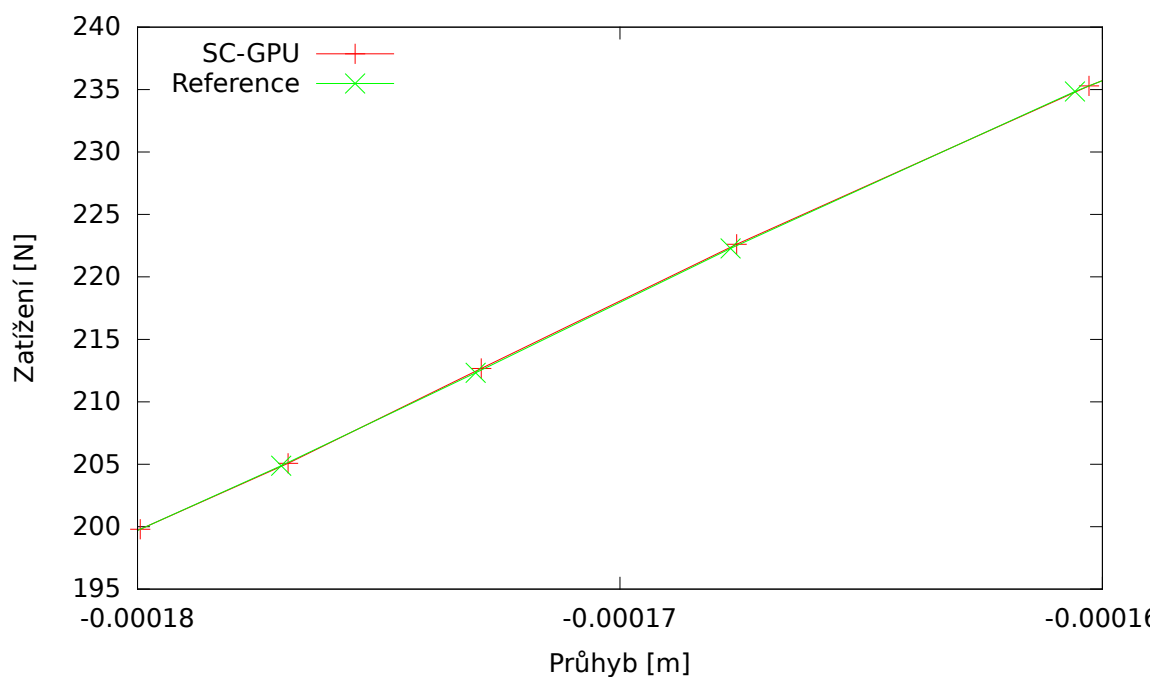


Obrázek 7.6: Času jedné simulace na při zapnutém a vypnutém Hyper-Threading (HT) na jednom a dvou socketech.

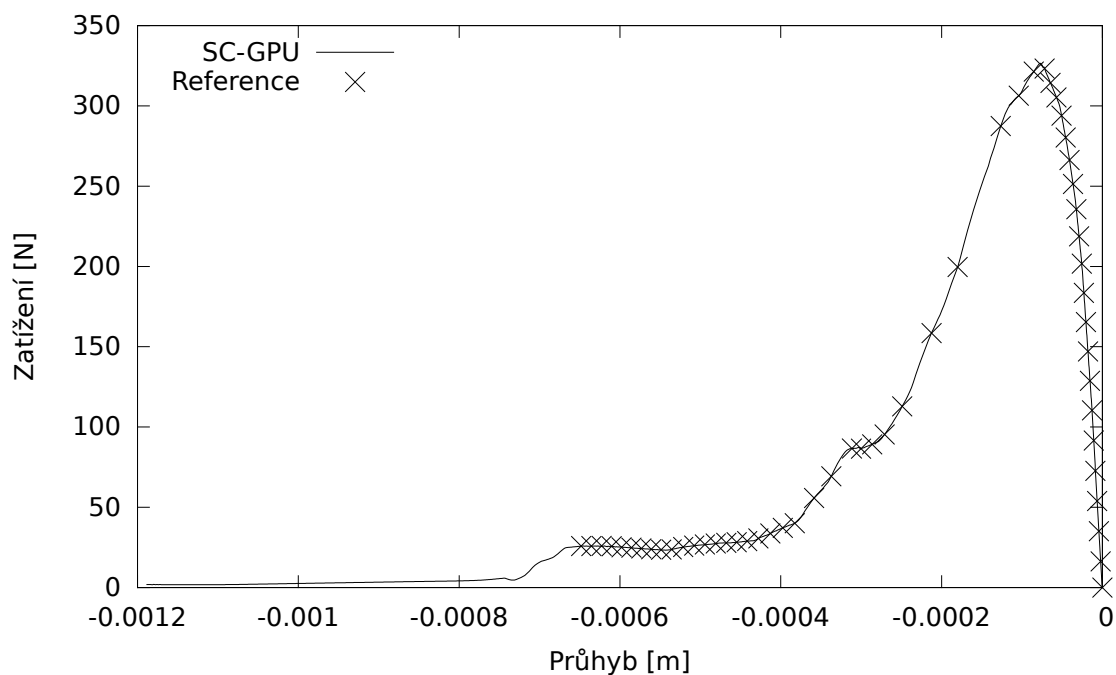


Obrázek 7.7: Změny ve výkonu na jednom a dvou procesorech se zapnutou a vypnutou funkcí Hyper-Threading (HT).

Nárůst výkonu na jednom procesoru není velký, jelikož jedno vlákno využívá stejné prostředky jako druhé, a tím nedovoluje druhému vláknu běžet současně. Propad u dvou-socketové verze lze přičíst dvojnásobné komunikaci mezi procesory.



Obrázek 7.8: Detail výsledného grafu průhybu trávce vůči zatížení. Body jsou sice posunuty oproti referenčním, ale tvoří stejnou křivku s minimální odchylkou.



Obrázek 7.9: Zobrazení výsledného grafu a referenčního výsledku z prototypu. Výsledný graf prochází body prototypu. (Pro přehlednost byl v grafu použit pouze každý desátý bod referenčního výstupu.)

## 7.6 Přesnost výsledků

Některé techniky akcelerace výpočtu umožňují rychlejší běh programu za cenu méně přesných výsledků, viz Kapitola 6.3. Měření přesnosti výsledného grafu vůči výsledku prototypu nebylo možné automatizovat. Výsledné body neodpovídaly referenčním bodům a byly různě na křivce posunuty. Ležely však na křivce dané výstupem prototypu. Kontrola byla vždy prováděna vizuálně. Výsledná verze produkuje na svém výstupu shodnou křivku.

## 7.7 CUDA realizace

Jedním z přání Ústavu stavební mechaniky Fakulty stavební bylo, aby šel kód spustit na grafické kartě. Ústav má k dispozici grafické karty Nvidia Tesla C2050 a C1060 a chtěl by je k řešení simulace využít. Tyto karty mají potenciálně vysoký výkon pro zpracování paralelních dat. Úloha simulace popsaná v této práci však využití tohoto potenciálu do jisté míry limituje.

Grafické karty se přednostně věnují výpočtům v plovoucí řádové čárce s jednoduchou předností, tedy *float*. V závislosti na použité architektuře procesoru grafické karty je výkon v *double* výrazně nižší. U C2050 je výkon v *double* poloviční oproti *float* <sup>1</sup>. U C1060 <sup>2</sup> je tento poměr daleko horší. Nejvyšší teoretický výkon ve *float* je 933 GFlops, kdežto ve *double* pouhých 78 GFlops. Tohoto teoretického výkonu je obtížné dosáhnout, říká však, že za čas zpracování dvanácti *float* hodnot dokáže C1060 zpracovat jednu hodnotu ve *double*.

Dalším problémem je implementace funkce počítající reakce působící na *brick* elementy. CUDA zpracovává vždy 32 vláken současně. Těchto 32 vláken je označováno jako WARP a je potřeba, aby vlákna v rámci jednoho WARPU vykonávala stejnou práci, tedy pracovala synchronně a přistupovala do paměti společně. Při ideální situaci načítá 32 sousedních vláken 32 v paměti sousedících hodnot. Přístup k bodům v rámci *brick* však není zarovnán. První dvě vlákna/*brick* mohou sdílet čtyři body, další dvě pouze jeden. Tento přístupový vzor do paměti není na grafické kartě příliš vhodný, jelikož tyto přístupy zpomalují přípravu dat do WARPU.

Přístup do paměti je problematický i při ukládání dat. Jelikož jsou reakce z více *brick* sčítány dohromady, je potřeba, aby toto přičítání provádělo vždy jedno vlákno. Jinak dochází k soutěžení. Operace přičítání do paměti spočívá v načtení hodnoty, její zvýšení o požadovanou hodnotu a následné uložení. Dvě vlákna mohou najednou přičíst hodnotu, zvýšit ji o svou a současně uložit. V paměti tak bude výsledek jednoho nebo druhého vlákna, ne však jejich součet. Při atomickém zápisu k paměti dochází k serializaci výpočtu a namísto masivní paralelizace, které je grafická karta schopna, jsou data zpracovávána takřka sekvencně.

Limitaci možností běhu na grafické kartě představuje i režie spuštění kernelu. Vzhledem k rozdílným přístupovým vzorům je potřeba spustit těchto kernelů více. Výpočet změn vzdálenosti, reakci uvnitř *brick*, nových pozic bodů a kohezivní trhliny dávají alespoň 4 samostatné postupně spouštěné kernely. Optimalizovaný čas na iteraci je 132 $\mu$ s. Režie spuštění prázdného kernelu se pak pohybuje kolem 10 $\mu$ s [11]. Zbylý čas na výpočet jedné iterace po odečtení předpokládané režie čtyř kernelů by měl být menší než 90 $\mu$ s. Je otázkou, zda přenesený výpočet na grafickou kartu bude moci běžet rychleji než paralelní CPU verze.

V době psaní této práce není CUDA realizace dokončena. Princip výpočtu a uložení dat se ukázaly jako problematické pro běh na grafické kartě. Je potřeba provést přeskládání

<sup>1</sup>[http://www.nvidia.co.uk/object/product\\_tesla\\_C2050\\_C2070\\_uk.html](http://www.nvidia.co.uk/object/product_tesla_C2050_C2070_uk.html)

<sup>2</sup>[http://www.nvidia.co.uk/object/tesla\\_c1060\\_uk.html](http://www.nvidia.co.uk/object/tesla_c1060_uk.html)

dat v paměti tak, aby s nimi mohla grafická karta lépe pracovat. Dokončení této realizace je plánováno na konec června a začátek července 2017.

## Kapitola 8

# Závěr

Tato práce rozebírá problém simulace lomu kvazikřehkých materiálu na počítačových systémech. Použitá metoda a její implementace je zde analyzována z principu matematického výpočtu kvazikřehkého lomu i z pohledu efektivity využití výpočetních prostředků. Pro existující implementaci simulace byly navrženy úpravy umožňující zkrácení běhu simulace, a to při spuštění simulace na jednom jádře procesoru i na několika jádrech.

Analýza původní implementace odhalila četné provádění skalárních operací nad proměnnými datového typu *double*. Přesněji 66 procent běhu simulace na jednom jádře připadalo na jeden řádek provádějící v cyklu součin matic. S pomocí rozhraní OpenMP bylo toto skalární násobení převedeno do vektorizačních AVX jednotek procesoru. Spolu s metodou rozbalení smyček bylo možné urychlit výpočet na jednom vlákne 3,69 krát.

Taktéž s pomocí rozhraní OpenMP byla provedena paralelizace zpracování konečných elementů. Na architektuře s jednotnou dobou přístupu do paměti, tedy jednoprocesorovém systému, s osmi jádry bylo dosaženo dalšího dalšího 5,7 krát rychlejšího běhu. Při použití všech osmi jader je tak výsledná verze 21 krát rychlejší než původní prototyp. Účinnost, efektivita této výsledné paralelní verze je 71,25 procent.

Při použití dvou procesorů na jedné základní desce a rozhraní OpenMP poklesla efektivita řešení na 24,68 procent. Tento pokles je způsoben krátkými iteracemi a značnou režii komunikace mezi procesory.

Analýzou stupňováním frekvence bylo zjištěno, že v rozmezí frekvencí 1,2 GHz až 2,4 GHz není výsledná paralelní verze bržděna přístupovou dobou k paměti. Procesory s vyšší frekvencí by tak mohly poskytnout kratší čas simulace, než v práci prezentovaný 1 320 sekund na osmijádrovém procesoru.

Byl také ověřen vliv technologie Hyper-Threading na výkon aplikace. U jednosoketové paralelní verze byl naměřen 3 procentní nárůst výkonu počtu iterací za sekundu pokud při zapnutí Hyper-threading. U dvousoketové verze byl pak naopak zaznamenán 33 procentní propad tohoto výkonu.

Na základě této práce vznikl článek pro konferenci Excel@FIT 2017<sup>1</sup>, který byl zde prezentován formou plakátu. S odbornou veřejností byly diskutovány použité techniky optimalizace i další možnosti jako například běh simulace na výpočetních akcelerátorech Intel Xeon Phi nebo systémech na čipu Nvidia Tegra.

V rámci práce byla také zkoumána možnost využití k výpočtu grafické akcelerátory pomocí jazyka CUDA. Toto řešení se během práce na tématu nepodařilo dokončit. Současný CUDA prototyp využíván na Ústavu stavební mechaniky zpracovává referenční model 100

---

<sup>1</sup><http://excel.fit.vutbr.cz/submissions/2017/036/36.pdf>



minut. V této práci prezentovaná sekvenční verze zpracovává stejné množství práce za 120 minut. Paralelní verze pak za 21 minut. Jestli bude v možnostech grafické karty zpracovávat jednotlivé iterace rychleji než zde prezentovaný procesor zůstává otázkou. Práce na CUDA verzi bude pokračovat v letních měsících roku 2017.

# Literatura

- [1] Allinea MAP - C/C++ profiler and Fortran profiler for high performance Linux code. [Online; navštíveno 10.5.2017].  
URL <https://www.allinea.com/products/map>
- [2] GCC, the GNU Compiler Collection. [Online; navštíveno 10.5.2017].  
URL <https://gcc.gnu.org>
- [3] Intel® 64 and IA-32 Architectures Optimization Reference Manual. [Online; navštíveno 10.5.2017].  
URL <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf>
- [4] Intel® C++ Compilers. [Online; navštíveno 10.5.2017].  
URL <https://software.intel.com/en-us/c-compilers>
- [5] Memory Layout Transformations. [Online; navštíveno 9.5.2017].  
URL <https://software.intel.com/en-us/articles/memory-layout-transformations>
- [6] MPI Forum. [Online; navštíveno 11.5.2017].  
URL <http://mpi-forum.org/>
- [7] The OpenMP API specification for parallel programming. [Online; navštíveno 11.5.2017].  
URL <http://www.openmp.org/>
- [8] PAPI. [Online; navštíveno 9.5.2017].  
URL <http://icl.cs.utk.edu/papi/>
- [9] Ahmdal, G. M.: Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings (30)*, 1967: s. 483–485, DOI: 10.1145/1465482.1465560.  
URL <http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf>
- [10] Bedáň, J.: Simulace lomových procesů na superpočítači. *Pojednání k tématu disertační práce, FAST VUT v Brně*, 2015.
- [11] Boyer, M.: LAVA Lab CUDA Support. [Online; navštíveno 10.5.2017].  
URL [http://www.cs.virginia.edu/~mwb7w/cuda\\_support/](http://www.cs.virginia.edu/~mwb7w/cuda_support/)
- [12] Frantík, P.: Jednoduchý model lomu trámce. [Online; navštíveno 28.12.2016].  
URL <http://www.kitnarf.cz/publications/2004/2004.06.plm4/2004.06.plm4.html>

- [13] Griffith, A.: The Phenomena of Rupture and Flow in Solids. *Philosophical Transactionsm, Series A, Vol. 221*, 1920: s. 163–198.
- [14] Gustafson, J. L.: Reevaluating Amdahl’s Law. *Communications of the ACM. 31 (5)*, 1988: s. 532–533, DOI: 10.1145/42411.42415.  
URL <http://www.johngustafson.net/pubs/pub13/amdahl.htm>
- [15] Lomont, C.: Introduction to Intel Advanced Vector Extension. [Online; navštíveno 9.5.2017].  
URL [https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro\\_to\\_Intel\\_AVX.pdf](https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf)
- [16] Molka, D.; Hackenberg, D.; Schöne, R.: Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer. *MSPC '14, June 13, 2014, Scotland*., 2014: str. 4.
- [17] Moore, G. E.: Cramming more components onto integrated circuits. *Electronics, Vol. 38, No. 8.*, 1965: s. 114–117, DOI: 10.1109/jproc.1998.658762.  
URL <http://ieeexplore.ieee.org/document/658762/>
- [18] Novák, D.; Brdečko, L.: *Pružnost a pevnost MO1 - Základní pojmy a předpoklady*. FAST VUT v Brně, 2004, 27-28 s.
- [19] Pail, T.; Frantík, P.; Štafa, M.: Specializovaný MKP model lomu trámce. *International Scientific Conference on Structural and Physical Aspects of Civil Engineering*, Slovenská republika: 2010: s. 1–6, ISBN: 978-80-89284-68-9.
- [20] Society, I. C.: IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, 2008: s. 1–70, DOI: 10.1109/IEEESTD.2008.4610935.

# Přílohy

## Seznam příloh

### A Obsah přiloženého paměťového média

50

## Příloha A

# Obsah přiloženého paměťového média

DVD přiložené k této práci obsahuje následující strukturu souboru a složek:

- **/repozitar.tar.gz** Exportovaný repozitář z GitLab,
- **/src/** Zdrojové soubory výsledné CPU verze simulace,
- **/doc/** Zdrojové soubory k Diplomové práci, Semestrálnímu projektu a článku pro Excel@FIT 2017,
- **/models/** Soubor vstupních modelů simulace,
- **/output/** Výstupní soubory experimentu se simulací v různých částech optimalizace,
- **/README.md** Informace o struktuře projektu, požadavcích na spuštění, návod na kompilaci a spuštění,
- **/Makefile** Makefile pro překlad simulace.